
Evaluación y aceleración del algoritmo Path tracing en arquitecturas heterogéneas

Por
Enrique de la Calle Montilla



UNIVERSIDAD COMPLUTENSE MADRID

Grado de Ingeniería Informática
FACULTAD DE INFORMÁTICA

Carlos García Sánchez
**Evaluation and acceleration of Path Tracing
algorithm in heterogeneous architectures**

MADRID, 2020-2021

Índice general

1. Introducción	8
1.1. Objetivos	8
1.2. Plan de trabajo	9
1.2.1. Trabajo previo	9
1.2.2. Desarrollo del trabajo	9
2. Implementación básica de Path Tracing	11
2.1. Esquema Path Tracing	11
2.1.1. Aproximación de la ecuación de renderizado	11
2.1.2. Esquema de resolución	12
2.2. Pipeline	13
2.2.1. Preparado de escena	13
2.3. Acumulado de muestras	16
2.3.1. Renderizado Progresivo	16
2.4. Trazado de rayo desde la cámara	17
2.5. Intersección en la escena	18
2.5.1. Intersección triángulo - rayo	18
2.5.2. Kernel principal	20
3. Mejoras visuales	23
3.1. Desenfoque: Modelo de lente fina	23
3.2. Texturas	25
3.2.1. Mapeo	25
3.2.2. IBL (Image Based Lighting)	26
3.2.3. Filtrado	28
3.2.4. Mapas de normales	29
3.3. Smooth Shading	30
3.4. Sombreado BRDF	32
4. Optimizaciones estructurales	34
4.1. Muestreo por importancia	34
4.1.1. Muestreo por importancia de Estimación de Evento Próximo (NEE)	35
4.1.2. Muestreo por importancia de entorno	36
4.1.3. Muestreo por importancia de BRDF	40
4.1.4. Muestreo por importancia múltiple	40
4.2. Estructuras de aceleración	41
4.2.1. Operación de generación de BVH (CPU)	42
Partición por plano:	44
Heurística SAH:	45

4.2.2. Operación de recorrido (GPU)	45
Comparación de heurísticas:	48
4.2.3. Evaluación:	49
5. Evaluación	51
5.1. Análisis Roofline	52
6. Port oneAPI	53
7. Conclusiones	54
7.1. Trabajo Futuro	55
7.2. Future work	56
8. Anexo	57
8.1. Manual	57
8.1.1. Formato de escenas	57
8.2. Galería	58

Resumen

En este trabajo se explica la implementación de Eleven Renderer, un motor de renderizado gráfico programado en C++ y CUDA de código abierto. Cuenta con todas las características necesarias para visualizar una escena 3D con relativa eficiencia y que hace uso de la arquitectura paralela que ofrecen los aceleradores gráficos. Además se acompaña con una evaluación y justificación de los detalles de la implementación además de ofrecer recursos en caso de querer ampliar información. Así pues, se espera que este trabajo sirva como breve guía y motivación para futuros programadores gráficos.

Eleven Renderer permite importar escenas desde la gran mayoría de software de edición 3D, aunque requiere de un proceso manual para adaptar el formato de la escena. Cuenta con una versión en oneAPI que puede ejecutarse en distintas plataformas sin requerir el uso de tarjetas gráficas de NVIDIA.

Abstract

Agradecimientos

Eleven Renderer



Capítulo 1

Introducción

El renderizado 3D es una técnica por la cual se representan los datos de geometrías tridimensionales en una pantalla en dos dimensiones. Existen distintas técnicas de renderizado tridimensional, siendo el trazado de rayos la más utilizada en la industria cinematográfica gracias a su excepcional fotorrealismo del cual suelen carecer otras técnicas. Tan codiciada es esta técnica que, actualmente industrias relacionadas con la visualización 3D como por ejemplo la industria de los videojuegos trata de replicarla y adaptarla en sus proyectos más modernos, con el fin de obtener mayor fidelidad visual. El creciente interés ante el trazado de rayos ha provocado que compañías dedicadas al hardware gráfico dediquen parte de su desarrollo a la implementación del trazado de rayos en sus últimos modelos. Véase, por ejemplo, las últimas dos series de tarjetas gráficas de NVIDIA, las series RTX (del inglés Ray Tracing Texel eXtreme).

No obstante, la implementación de este sistema de renderizado peca de una gran demanda computacional, por lo que gran parte de los motores de renderizado orientados a la industria cinematográfica han sido adaptados para poder funcionar en aceleradores gráficos. Con la creciente capacidad computacional y las características fotorrealistas mencionadas anteriormente, los motores de renderizado de trazado de rayos han adquirido una especial importancia en el ámbito de la computación gráfica.

1.1. Objetivos

Este trabajo cubre brevemente las bases teóricas de un motor de renderizado fotorrealista con características a la orden del día de otros motores comerciales, además de haber realizado una implementación en CUDA. Además, este trabajo hace hincapié en el análisis de dicha implementación en GPUs modernas, poniendo a prueba distintos parámetros y optimizaciones.

La implementación cuenta con características cercanas al estado del arte: es capaz de renderizar escenas del orden de millones de triángulos en tiempos razonables; usa un modelo de sombreado realista desarrollado por Disney [1]; permite el uso de mapas de textura y además funciona en GPU.

Un aspecto importante en cuanto a la implementación es que se hace uso del menor número de librerías posibles, con el fin de indagar a fondo en la arquitectura de un motor de renderizado de Path Tracing de manera íntegra. Así pues, se desmitifica el trabajo que normalmente se delega a las librerías gráficas, y se expone de manera clara el funcionamiento de cada pieza de un motor de renderizado de Path Tracing en GPU.

1.2. Plan de trabajo

1.2.1. Trabajo previo

Actualmente existen diversas implementaciones de motores de renderizado por Path Tracing de código abierto. La mayoría de ellas son implementaciones directas de los libros Ray Tracing in one Weekend [13] o del libro Physically based rendering: From theory to implementation [11] siendo el primero una rápida implementación funcional y mientras que el segundo contiene casi todos los fundamentos teóricos acompañados de una implementación muy completa. Aún así estos libros están orientados a renderizado por CPU, toda la arquitectura que acompaña sobre todo el segundo libro busca el mayor nivel de abstracción posible, esto es un problema a la hora de usar CUDA, que no permite el uso de funciones virtuales y por tanto cuenta con una herencia de clases limitada.

La mayoría de estas implementaciones son pruebas de concepto como la que se desarrolló en el trabajo final de la asignatura "Programación de GPUs y aceleradores". Estas pruebas de concepto suelen estar limitadas en cuanto a características, pero hay una implementación concreta que destaca: Cycles.

Cycles es sin duda una de las implementaciones de código abierto más relevantes de este algoritmo. Es un motor de renderizado desarrollado por Blender, flexible y con implementación en distintas APIs para poder hacer uso de múltiples tipos de aceleradores, entre ellas CUDA. Además ofrece una amplia gama de características como motor gráfico, a la par de las características vistas en el estado del arte, que le permiten hacerse hueco entre el resto de motores de render de producción comerciales.

Hasta ahora se ha hablado de las implementaciones de código abierto, pero es imposible ignorar las aplicaciones comerciales. Debido a que el renderizado fotorrealista es una herramienta muy cotizada en la industria audiovisual, grandes empresas han desarrollado o adquirido motores de Path Tracing propietarios. Unos cuantos ejemplos de motores de renderizado comerciales acelerados por GPU se muestran a continuación:

- Autodesk: Arnold Render
- Blender Foundation: Cycles
- Marmoset: Marmoset Toolbag
- Otoy: Octane Render
- Maxon: Redshift
- Pixar: Renderman
- Chaos Group: V-Ray

La gran mayoría de los motores mencionados funcionan exclusivamente con aceleradores de NVIDIA debido a que están implementados con CUDA. De la misma manera, Eleven Renderer hará uso de esta misma API.

1.2.2. Desarrollo del trabajo

El objetivo del desarrollo de este motor de render ha sido implementar el mayor número de características de un motor gráfico de producción y analizarlas una a una. Es por ello que la planificación ha carecido de una estructura formal, y ha sido remodelada constantemente a partir de los logros y el progreso obtenido hasta la fecha.

CUADRO 1.1 Timeline

Octubre 2020	Finalización de lectura del curso TU WIEN Rendering
Noviembre 2020	Inicialización de proyecto, prototipo inicial de trazado de rayos en CUDA y visualización con SFML, implementación de texturas
Diciembre 2020	Mapeado UV, Disney BRDF, carga de modelos .obj
Enero 2021	BVH construcción y recorrido
Abril 2021	SAH y comienzo de escritura de la memoria, visualizador BRDF
Febrero 2021	HDRI y NEE
Marzo 2021	Muestreo por importancia de HDRI
Mayo 2021	Funciones de benchmarking
Julio 2021	Port One Api
Agosto 2021	Implementación de mapas de normales, BOKEH
Septiembre 2021	Escenas en formato JSON

A continuación se muestra una línea del tiempo con los hitos logrados:

Cabe destacar que toda figura es original y que todas las muestras y comparaciones visuales de renderizado han sido realizadas con Eleven Renderer salvo excepciones que están debidamente señaladas.

Capítulo 2

Implementación básica de Path Tracing

En este capítulo se procede a dar un esquema básico de los fundamentos de este algoritmo. El objetivo será así describir un motor de renderizado previo a cualquier optimización, que cuenta con las funcionalidades básicas para producir una imagen de una escena tridimensional simple.

2.1. Esquema Path Tracing

2.1.1. Aproximación de la ecuación de renderizado

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\omega_i \cdot \mathbf{n}) d\omega_i$$

La Ecuación de Renderizado [4] aparece por primera vez en 1986 junto al algoritmo de Path Tracing, siendo este algoritmo una propuesta para resolverla. Es el pilar de la visualización 3D fotorrealista, ya que simula de una manera suficientemente precisa la interacción de la luz en una escena tridimensional. En este trabajo se utiliza una interpretación más moderna que sustituye términos como el "término geométrico" y adapta dicha ecuación al estado del arte.

La interpretación de esta ecuación es la siguiente: Para un punto \mathbf{x} del espacio y un ángulo ω_o desde el cual se observa a dicho punto, cuál es la cantidad de energía lumínica que el observador recibe L_o . La interpretación geométrica se muestra en la Figura 2.1.

El primer término $L_e(\mathbf{x}, \omega_o)$ indica la luz que dicho punto \mathbf{x} emite, así pues se podrán modelar materiales que emitan luz propia y no dependan de energía externa.

El segundo término calcula toda la luz entrante y reflejada a través del ángulo ω_o por dicho punto \mathbf{x} , es por ello que integra todos los ángulos del hemisferio superior. Este segundo término se compone de tres coeficientes:

El primer coeficiente $f_r(\mathbf{x}, \omega_i, \omega_o)$ es la función BRDF, la cual es dependiente del material e indica cuánta energía se refleja en dicho punto para las direcciones de entrada ω_i y salida ω_o .

El segundo coeficiente $L_i(\mathbf{x}, \omega_i)$ hace referencia a toda la energía lumínica entrante de todas las direcciones posibles.

El tercer coeficiente $(\omega_i \cdot \mathbf{n})$ es el producto de la ley del coseno de Lambert [7], un escalar que atenúa los ángulos menos pronunciados con la normal de la superficie.

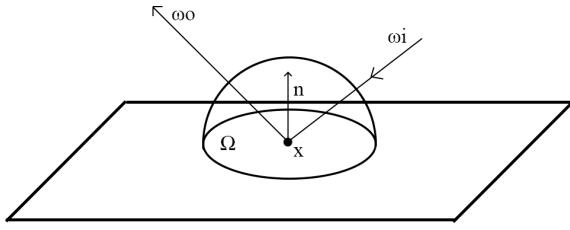


Figura 2.1: Interpretación geométrica del término de integración de la ecuación de renderizado

2.1.2. Esquema de resolución

Kajiya [4] propone junto a la ecuación el algoritmo de Path Tracing para resolverla. Este algoritmo se fundamenta en el método de Monte Carlo debido a la imposibilidad de resolver dicha ecuación de manera analítica. Para aproximar una integral con el método de Monte Carlo, basta con muestrear aleatoriamente la función, y tras varias muestras es posible aproximar dicha integral. En la práctica, estas muestras son rayos con direcciones aleatorias y se tratará de aproximar la ecuación de renderizado.

$$\int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\omega_i \cdot \mathbf{n}) d\omega_i \approx \frac{1}{N} \sum_{k=1}^N \frac{f_r(\mathbf{x}, \omega_k, \omega_o) L_k(\mathbf{x}, \omega_k) (\omega_k \cdot \mathbf{n}) d\omega_k}{p(\omega_k)}$$

Así pues el procedimiento general es el siguiente:

1. Calcular un rayo inicial desde la posición de la cámara a cada píxel del sensor de la cámara.
2. "Lanzar" dicho rayo a la escena.
3. Comprobar la intersección de este rayo con el elemento más cercano al origen de este. En caso de no intersecar se entiende que el rayo ha salido de la escena, de manera que se añade la luz del fondo a dicho camino y se salta al punto.
4. Aplicar la reducción de energía pertinente a dicha intersección.
5. Comprobar si se ha alcanzado el número máximo de rebotes, en caso negativo se calcula la dirección a la que rebota dicho rayo y se vuelve al paso 2.
6. Se añade al píxel la luz que contiene dicho rayo.

Este procedimiento queda más claro en la Figura 2.2 donde se utilizan los nombres de las funciones utilizadas en la implementación.

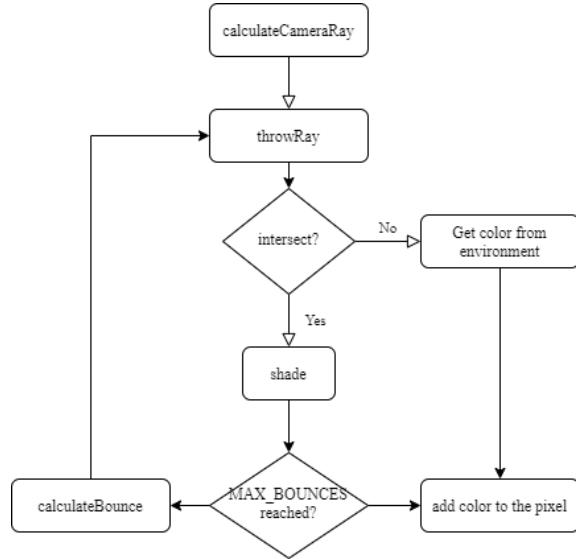


Figura 2.2: Esquema Path Tracing

2.2. Pipeline

En primer lugar es apropiado dar una visión de la estructura que se ha programado para Eleven Rendererer. Esta estructura está formada por una serie de clases y procedimientos cuyo objetivo es simular una escena 3D, transferirla a la GPU, ejecutar el algoritmo, transferir de vuelta el resultado como una imagen final y por último mostrar esta imagen al usuario. Además se ofrecerán distintas métricas con el fin de evaluar y conocer el estado del algoritmo para determinada escena.

2.2.1. Preparado de escena

El primer paso es preparar la escena a renderizar `Scene`. Una escena básica se compone de una cámara `Camera`, geometrías `MeshObject` y materiales `Material`.

La clase `Scene` contiene un array dinámico para cada conjunto de elementos, con sus pertinentes funciones de añadido. Por ejemplo, la función `addMeshObject()` lleva la cuenta de objetos en la escena y actualiza el ID de cada objeto a partir de esta cuenta. Además esta clase cuenta con una función denominada `Scene sceneBuilder(std::string path)` que cargará una escena localizada en un directorio. Más detalles sobre el formato de estas escenas se puede encontrar en el manual ubicado en el anexo

Los elementos básicos que almacena una escena son los siguientes:

- Cámaras `Camera`: consisten en una simulación aproximada de una cámara física real, así pues sus atributos serán: tamaño del sensor (en metros) `sensorWidth` y `sensorHeight`, distancia focal (en metros) `focalLength` y resolución (en píxeles) `xRes` e `yRes`.
- Geometrías `MeshObject`: son elementos que contienen un conjunto de triángulos `Tri` los cuales a su vez consisten en 3 puntos tridimensionales `Vector3 vertices[3]`.
- Materiales `Material`: definen la manera en la que los fotones interactúan con ellos. En su forma más primitiva consisten en un color base, el cual absorberá ciertas longitudes de onda en mayor o menor medida. Para simplificar las computaciones, no

es necesario calcular estas interacciones con todo el espectro electromagnético visible, basta con usar los tres colores primarios aditivos: rojo, verde y azul. Así, un color consiste en un vector `Vector3(R,G,B)`.

Una vez preparada la escena se inicia el proceso de renderizado. La función `startRender` prepara dos buffers en la CPU, `rawPixelBuffer` que recibirá los píxeles procedentes del algoritmo, previos a cualquier modificación y `beautyBuffer` que contendrá la imagen final preparada para ser mostrada en pantalla.

El siguiente paso es transferir la escena a la GPU, donde se realizará el resto de computaciones más demandantes. Esto es realizado por la función `renderSetup`, que a través de las funciones de la API de CUDA `cudaMalloc` y `cudaMemcpy` copia la información de la escena y lleva la cuenta de la memoria transferida en las variables globales `textureMemory`, `geometryMemory`. La copia de memoria de la CPU a la GPU requiere de un tratado especial en cuanto a objetos con atributos refiere, requiriendo así una copia profunda de los atributos en formato array o punteros, además de un proceso posterior denominado "pointer binding". Este proceso como se aprecia en la Figura 2.3 es necesario puesto que los objetos que hacen referencias a punteros al ser copiados a la GPU pierden dicha referencia al cambiar de contexto.

Para ello si un objeto `Object` contiene un atributo `int* array`, habrá que asignar espacios de memoria diferentes para cada uno, copiar ambos independientemente y posteriormente, a través de la función `cudaMemcpy`, copiar la dirección del array a la dirección del atributo.

```
cudaMemcpy(&(Object->array), &(array), sizeof(array*), cudaMemcpyHostToDevice);
```

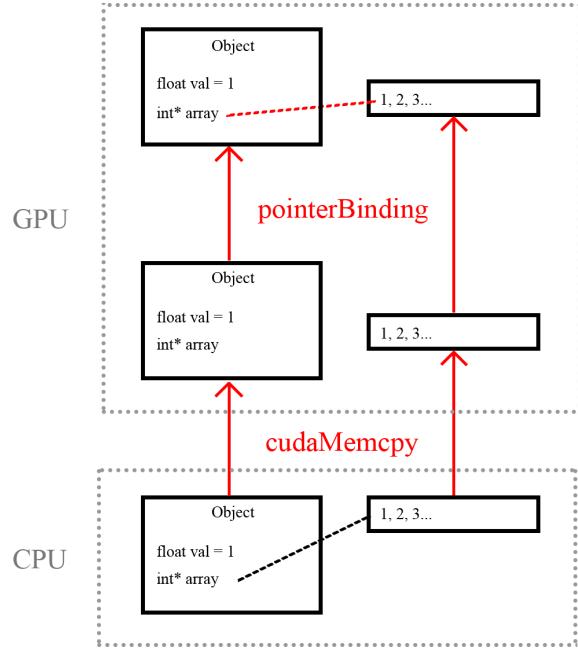


Figura 2.3: Pointer Binding

La función `startRender`, tras realizar toda la copia de los elementos a la GPU, ejecuta un kernel que configura algunos parámetros iniciales. `setupKernel` cuyo código se muestra en Listing 2.1, se ejecuta de manera síncrona y se hace cargo de inicializar los buffers de píxeles y contador de rayos, inicializar `curand` (la librería utilizada para generar números aleatorios en CUDA) y por último, enlazar cada `MeshObject` con sus pertinentes triángulos.

Esto es llevado a cabo con un puntero a una posición de la lista global de triángulos, ya que por motivos que posteriormente se explican en Estructuras de Aceleración es conveniente almacenar todos los triángulos por separado.

Listing 2.1: Kernel de configuración inicial

```
--global__ void setupKernel() {

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    int idx = (dev_scene_g->camera->xRes * y + x);

    if ((x >= dev_scene_g->camera->xRes) || (y >=
        dev_scene_g->camera->yRes)) return;

    dev_samples[idx] = 0;
    dev_pathcount[idx] = 0;

    cudaMemset(&dev_buffer[4 * idx], 0, sizeof(float) * 3); // RGB
    channels
    dev_buffer[4 * idx + 3] = 1; // Alpha channel

    curand_init(0, idx, 0, &d_rand_state_g[idx]);

    // Just one thread
    if (x == 0 && y == 0) {
        int triSum = 0;
        for (int i = 0; i < dev_scene_g->meshObjectCount; i++) {
            dev_scene_g->meshObjects[i].tris += triSum;
            triSum += dev_scene_g->meshObjects[i].triCount;
        }
    }
}
```

Una vez la escena está correctamente configurada en la GPU, se crea un hilo nuevo que ejecutará en bucle el kernel de renderizado tantas iteraciones como muestras se hayan definido en los parámetros `RenderParameters`. Es necesario el uso de la librería `std::thread` para este proceso, ya que este bucle de llamadas a la GPU es un proceso bloqueante. El uso de dos hilos, uno para la ejecución de los kernels de renderizado y otro para la recolección de datos de la GPU de manera asíncrona, permite una previsualización dinámica del resultado del motor. Más detalles sobre este proceso se describen en el apartado Renderizado Progresivo.

En este punto, el algoritmo está ejecutándose en la GPU. Para mostrar el resultado se crea una ventana con la librería multiplataforma `SFML`. Esta librería permite mostrar en pantalla una imagen en formato RGB de 8 bits. Para poder adaptar el resultado del algoritmo a una imagen visible es necesario aplicar una serie de funciones de postprocesado. `getRenderData` se encarga de aplicar estas funciones y obtener una imagen postprocesada visible, también conocida en informática gráfica como "Beauty Pass".

Las funciones a ejecutar de postprocesado son las siguientes:

1. `flipY`: Debido a la configuración de la cámara, los píxeles son calculados de abajo a arriba, es necesario entonces invertirlos en el eje Y.

2. `clampPixels`: Es la forma más sencilla de mapeo de tonos, limitar los valores al rango [0-1].
3. `applysRGB`: Los monitores no trabajan en un espacio de color lineal, por eso es necesario aplicar la curva de corrección de gamma 2.2.
4. `HDRtoLDR`: Por último, es necesario convertir el array de píxeles de punto flotante a bytes. Ya que los valores en punto flotante están en [0,1], basta con multiplicar por 255 y hacer un posterior truncamiento.

Finalmente la librería de `SFML` se encarga de mostrar la imagen y superpone un texto con el número de muestras calculadas hasta el momento, obtenidas con `getSamples`.

2.3. Acumulado de muestras

Por cada píxel por el que se traza un camino es necesario añadir la radiación de dicho camino. Este píxel cuenta con radiación previa acumulada de las anteriores muestras.

$$newAccumulatedLight = \frac{oldAccumulatedLight * samples}{samples + 1} + \frac{newLight}{samples + 1}$$

Dicha radiación queda reflejada en cada píxel siguiendo el código mostrado en Listing 2.2 donde `dev_samples[]` es una variable global que almacena un contador con el número de muestras acumuladas hasta el momento para un píxel determinado y `dev_buffer[]` almacena la radiación total por píxel hasta el momento. Por otro lado la función `isnan` es necesaria para evitar errores numéricos, puesto que en determinadas escenas se han encontrado artefactos gráficos.

Listing 2.2: Código para la acumulación de luz por cada píxel

```
unsigned int sa = dev_samples[idx]; // sa is the number of samples
                                   accumulated for this pixel.

if (!isnan(light.x) && !isnan(light.y) && !isnan(light.z)) {

    if (sa > 0) {
        dev_buffer[4 * idx + 0] *= ((float)sa) / ((float)(sa + 1));
        dev_buffer[4 * idx + 1] *= ((float)sa) / ((float)(sa + 1));
        dev_buffer[4 * idx + 2] *= ((float)sa) / ((float)(sa + 1));
    }

    dev_buffer[4 * idx + 0] += light.x / ((float)sa + 1);
    dev_buffer[4 * idx + 1] += light.y / ((float)sa + 1);
    dev_buffer[4 * idx + 2] += light.z / ((float)sa + 1);

    dev_samples[idx]++;
}
```

2.3.1. Renderizado Progresivo

Una ventaja de los motores de renderizado más modernos es el renderizado progresivo. En este proceso, las muestras se van acumulando poco a poco a lo largo de la imagen

hasta que terminan por converger. Esto difiere de los motores de renderizado por CPU tradicionales, que acumulan las muestras en secciones locales y una vez que acumulan las suficientes, pasan a la siguiente sección. Este trabajo utiliza una implementación progresiva con el fin de estar más cerca del estado del arte.

Este tipo de implementación se beneficia de la copia de datos asíncrona de la GPU. Mientras el kernel se ejecuta, un flujo de datos secundario hace la copia del buffer de la GPU en la CPU, pudiendo así actualizar la visualización del resultado varias veces por segundo.

Este flujo de datos secundario se implementa con el tipo de datos `cudastream_t` de la API de CUDA. Son necesarios dos flujos, uno denominado `kernelStream` y otro denominado `bufferStream`. Los kernels de inicialización y renderizado corren en el primer flujo, mientras que la función que obtiene el buffer es ejecutada en el segundo flujo. El código Listing 2.3 muestra la copia asíncrona de los buffers a través de la función `cudaMemcpyFromSymbolAsync`. También se hace copia del buffer de la suma de rayos emitidos por pixel `pathcountBuffer` con el fin de realizar métricas de eficiencia.

Listing 2.3: Código para obtener los buffers de la GPU

```

cudaError_t getBuffer(float* pBuffer, int* pathcountBuffer, int
size) {

    cudaStreamCreate(&bufferStream);

    cudaError_t cudaStatus = cudaMemcpyFromSymbolAsync(p pBuffer,
        dev_pBuffer, size * sizeof(float) * 4, 0, cudaMemcpyDeviceToHost,
        bufferStream);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "returned error code %d after launching
            addKernel!\n", cudaStatus);
    }

    cudaStatus = cudaMemcpyFromSymbolAsync(pathcountBuffer,
        dev_pathcount, size * sizeof(unsigned int), 0,
        cudaMemcpyDeviceToHost, bufferStream);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "returned error code %d after launching
            addKernel!\n", cudaStatus);
    }

    return cudaStatus;
}

```

2.4. Trazado de rayo desde la cámara

Para simular el trazado del rayo desde la cámara hasta la escena, se simula de manera simplificada como haría una cámara estenopeica. Para cada píxel del sensor, se calcula su posición en el espacio a partir del tamaño del sensor. Esto quiere decir que el píxel (0,0) se encuentra en la esquina inferior izquierda de la ubicación física del sensor y el píxel (`xRes`, `yRes`) se encuentra en la esquina superior derecha. Puesto que no se está teniendo en cuenta la rotación de la cámara, la coordenada `z` del sensor se puede simplificar con la distancia de la cámara hasta el sensor.

Si se observa un sensor físico real, se puede ver como los píxeles del sensor tienen un área. En la anterior explicación se determina la posición de cada píxel, pero es necesario muestrear a lo largo de todo el área del píxel para simular la realidad, no solo el centro de cada uno. Es por ello que se añade un término aleatorio `rx`, `ry` que se encargará de distribuir las posiciones a lo largo de la zona de cada píxel.

Listing 2.4: Trazado de rayo a cámara

```
__device__ void calculateCameraRay(int x, int y, Camera& camera, Ray&
    ray, float r1, float r2) {

    // Relative coordinates for the point where the first ray will be
    // launched
    float dx = camera.position.x + ((float)x) / ((float)camera.xRes) *
        camera.sensorWidth;
    float dy = camera.position.y + ((float)y) / ((float)camera.yRes) *
        camera.sensorHeight;

    // Absolute coordinates for the point where the first ray will be
    // launched
    float odx = (-camera.sensorWidth / 2.0) + dx;
    float ody = (-camera.sensorHeight / 2.0) + dy;

    // Random part of the sampling offset so we get antialiasing
    float rx = (1.0 / (float)camera.xRes) * (r1 - 0.5) *
        camera.sensorWidth;
    float ry = (1.0 / (float)camera.yRes) * (r2 - 0.5) *
        camera.sensorHeight;

    // Sensor point, the point where intersects the ray with the sensor
    float SPx = odx + rx;
    float SPy = ody + ry;
    float SPz = camera.position.z + camera.focalLength;

    // The initial ray is created from the camera position to the sensor
    // point. No rotation is taken into account.
    ray = Ray(camera.position, Vector3(SPx, SPy, SPz) - camera.position);
}
```

2.5. Intersección en la escena

Una vez se conoce el rayo primario que se lanza desde la cámara, es necesario determinar con qué objeto colisiona y la información de esa intersección. Esto se realiza por la función `throwRay` que, de forma ingenua, probará si el rayo intersecciona con cada triángulo de la escena y devolverá la intersección más cercana al origen de éste. En la Sección 4.2 se explica un método para evitar tener que probar a intersecar cada triángulo.

2.5.1. Intersección triángulo - rayo

El cálculo de la intersección de un rayo con un triángulo es una de las operaciones más fundamentales de este algoritmo. Esta operación toma como parámetros un triángulo `Tri` y un rayo `Ray` y ofrece como resultado si dicho triángulo y rayo intersecan en el espacio y

además un objeto `Hit` el cual cuenta con información adicional de la intersección.

La información adicional que devuelve esta operación es la siguiente:

- `int hit.objectID`: ID del objeto al que pertenece dicho triángulo.
- `Vector3 hit.position`: Posición en el espacio del punto de intersección entre el rayo y el triángulo.
- `Vector3 hit.normal`: Normal de la superficie, calculada a partir del producto vectorial de dos aristas del triángulo.
- `bool hit.valid`: Validez de una intersección, por defecto falso. Verdadero en caso de haber intersecado correctamente.

Para la implementación se ha hecho uso del algoritmo Fast Minimum Storage Ray-Triangle Intersection[10]. En este paper se explica detalladamente el algoritmo de intersección, mientras este trabajo se limita a implementar dicho algoritmo a partir de una adaptación de la implementación en C que el autor ofrece.

Listing 2.5: Adaptación de código Fast Minimum Storage Ray/Triangle Intersection[10] a Eleven Renderer

```

__host__ __device__ inline bool hit(Ray& ray, Hit& hit) {

    float EPSILON = 0.0000001;

    Vector3 edge1 = vertices[1] - vertices[0];
    Vector3 edge2 = vertices[2] - vertices[0];

    Vector3 pvec = Vector3::cross(ray.direction, edge2);

    float u, v, t, inv_det;

    float det = Vector3::dot(edge1, pvec);

    inv_det = 1.0 / det;

    if (det > -EPSILON && det < EPSILON) return false;

    Vector3 tvec = ray.origin - vertices[0];

    u = Vector3::dot(tvec, pvec) * inv_det;
    if (u < 0.0 || u > 1.0)
        return false;

    Vector3 qvec = Vector3::cross(tvec, edge1);
    v = Vector3::dot(ray.direction, qvec) * inv_det;
    if (v < 0.0 || (u + v) > 1.0)
        return false;

    t = Vector3::dot(edge2, qvec) * inv_det;

    if (t < 0) return false;

    Vector3 geomPosition = ray.origin + ray.direction * t;
    Vector3 geomNormal = Vector3::cross(edge1, edge2).normalized();

    hit.normal = geomNormal;
    hit.position = geomPosition;
    hit.valid = true;
    hit.objectID = objectID;

    return true;
}

```

2.5.2. Kernel principal

El kernel de renderizado `renderingKernel` se encarga de ejecutar el algoritmo descrito en la Figura 2.2, donde el número máximo de rebotes es una constante definida en tiempo de compilación como `MAX_BOUNCES`. La paralelización se hace a nivel de píxel, es decir, por cada bloque CUDA bidimensional, habrá una equivalencia a un grupo de píxeles del sensor. Este tipo de paralelización cuenta con la ventaja de ofrecer cierto grado de localidad

espacial en varios tipos de escena. Para escenas abiertas, la gran mayoría de los bloques no sufrirán del fenómeno de "branching", ya que la mayoría de caminos no intersecan con ningún objeto más que el cielo. Este método cuenta con limitaciones tras realizar un rebote aleatorio, es común que los propios hilos divergen ya que puede que algunos hilos realicen un solo rebote mientras que otros realicen MAX_BOUNCES - 1.

En la Figura ?? se muestra un mapa de calor para una escena abierta, realizado con Eleven Renderer. Los distintos colores indican el número de rebotes necesarios para cada píxel. Véase por ejemplo que las zonas de cielo cuentan con 0 rebotes debido a que no intersecan con nada, mientras que las partes más ocultas cuentan con un mayor número de rebotes. Esta breve visualización muestra como existe localidad bidimensional en el número de rebotes, de manera que justifica el enfoque de paralelización a nivel de píxel.

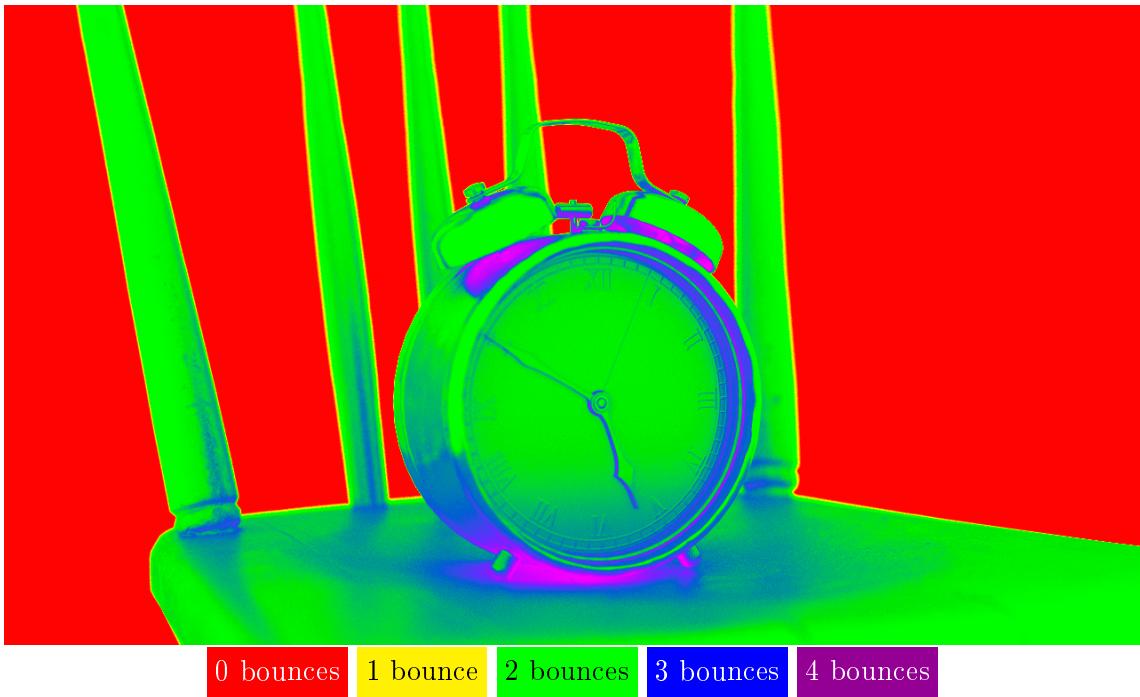


Figura 2.4: Mapa de calor para el número de rebotes de cada píxel

La ejecución de un kernel en CUDA debe estar acompañada de dos parámetros: el número de bloques y el número de hilos. En este caso, los bloques están distribuidos uniformemente a lo largo de la imagen como bloques bidimensionales de THREADSIZE * THREADSIZE píxeles. Estos parámetros se definen en Listing 2.6.

Listing 2.6: Selección de parámetros para el kernel principal

```

int tx = THREADSIZE;
int ty = THREADSIZE;

dim3 numBlocks(scene->camera.xRes / tx + 1, scene->camera.yRes / ty + 1);
dim3 threadsPerBlock(tx, ty);

```

Resulta interesante conocer el número de hilos óptimo por cada bloque, por lo que se han evaluado distintos tamaños de bloque en la Figura 2.5. En esta figura aparece por primera vez el término kPath/s. Como la eficiencia es un campo esencial en el renderizado

3D, Eleven renderer cuenta con unas pocas funciones que dan una breve información del estado actual del motor. Uno de estos datos es el número global de caminos que se trazan por segundo. La medida kPath/s es útil de manera comparativa para una misma escena, ya que indica el rendimiento general del motor. En el caso de la comparativa del número de hilos, aquellas ejecuciones de 8 hilos por dimensión, son las que han mostrado mayor número de caminos trazados por segundo, de manera que este es el número óptimo y el usado para el resto de la implementación.

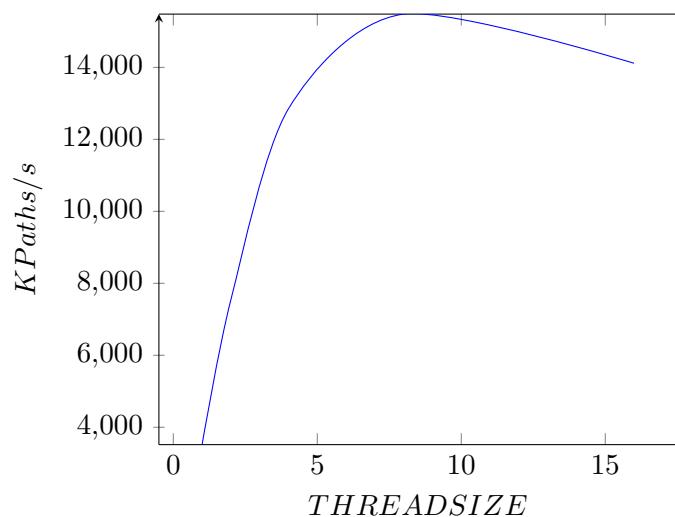


Figura 2.5: Comparación eficiencia con tamaño de bloque

Capítulo 3

Mejoras visuales

En el capítulo anterior se definió la implementación básica para visualizar una escena simple. Así pues este capítulo explica las mejoras visuales más comunes utilizadas en los motores gráficos de producción así como su implementación en Eleven Renderer y su evaluación.

3.1. Desenfoque: Modelo de lente fina

Hasta ahora se ha estado utilizando un modelo de cámara ideal denominado cámara estenopeica. Esta cámara tiene la particularidad de tener un enfoque perfecto siempre, siendo una propiedad indeseada en un motor de renderizado fotorrealista, ya que la mayoría de las cámaras reales incluyen lentes en su estructura que desvían los rayos de luz gracias a la difracción del cristal, enfocando a determinada distancia, y desenfocando el resto de la escena. El hecho de poder enfocar a una distancia determinada permite hacer énfasis en un sujeto de la escena, y desenfocar el resto. Este efecto se conoce como "Bokeh" y es muy deseado en un motor de renderizado, puesto que es un recurso cinematográfico muy atractivo visualmente.

Para solventar el problema del enfoque perfecto se hace uso de un modelo de cámara denominado modelo de lente fina. Este modelo es una simplificación de lo que sería una simulación física de unas lentes reales. Al simplificar los cálculos, se pierden artefactos y desperfectos deseados como la aberración cromática o la distorsión de lentes, pero a cambio se obtiene la simplicidad de implementación.

Para activar este efecto, es necesario compilar con la constante `BOKEH` definida. Esto desbloqueará la parte del código que hace el cálculo del desenfoque, ubicado en Listing 3.1

Este nuevo método añade a la clase `Camera` dos nuevas variables, por un lado `focusDistance` y por otro lado `aperture`. La primera define la distancia a la que se encuentra el plano de enfoque, y la segunda, la apertura en f-stops del iris de la cámara.

El procedimiento para calcular los rayos emitidos por el nuevo modelo de cámara se muestra de manera visual en la Listing 3.1 y es el siguiente:

1. Se calcula el rayo original del método anterior, desde la cámara hasta el sensor.
2. Se calcula la intersección de dicho rayo con el plano de enfoque, situado a la distancia `focusDistance`. La intersección se denomina `focusPoint`.
3. En vez de emitir el rayo desde el punto de la cámara, se elige un punto aleatorio en el iris `iRP` y se emite un rayo desde ahí hasta el punto de enfoque `focusPoint`. Este nuevo rayo será un rayo bajo el modelo de cámara de lente fina. Los elementos

situados a la distancia de enfoque `focusDistance` serán más nítidos que aquellos que no lo estén.

Listing 3.1: Código de desenfoque

```
#if BOKEH

float rIPx, rIPy;

// The diameter of the camera iris
float diameter = camera.focalLength / camera.aperture;

// Total length from the camera to the focus plane
float l = camera.focusDistance + camera.focalLength;

// The point from the initial ray which is actually in focus
Vector3 focusPoint = ray.origin + ray.direction * l;

// Sampling for the iris of the camera
uniformCircleSampling(r3, r4, r5, rIPx, rIPy);

rIPx *= diameter * 0.5;
rIPy *= diameter * 0.5;

Vector3 orig = camera.position + Vector3(rIPx, rIPy, 0);

//Blurred ray
ray = Ray(orig, focusPoint - orig);

#endif
```

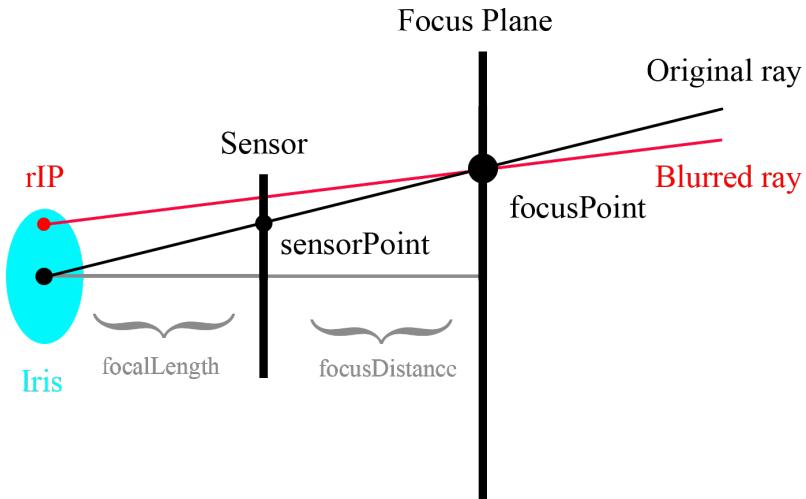


Figura 3.1: Esquema cámara modelo de lente fina

El resultado final es un desenfoque configurable a partir de la distancia de enfoque. Es posible también aumentar la cantidad de desenfoque reduciendo el número de f-stops, que consecuentemente aumentará la apertura del iris. La Figura ?? muestra un ejemplo de la modificación de distancia de enfoque mencionada.

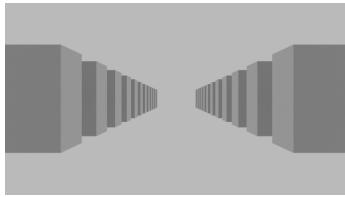


Figura 3.2: Desenfoque desactivado



Figura 3.3: Distancia de enfoque 10m

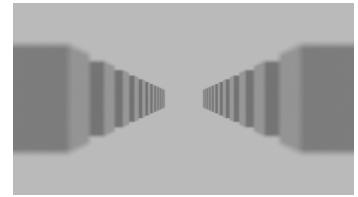


Figura 3.4: Distancia de enfoque 30m

3.2. Texturas

El uso de colores planos en los materiales limita la capacidad de imitación de la realidad. Un recurso esencial para romper esta limitación es el uso de texturas. Una textura consiste en una matriz bidimensional de valores en punto flotante. En la implementación se ha añadido una clase `Texture` que contiene dos valores enteros `width` y `height`, que indican la altura y anchura de la matriz de datos `data`. La carga se hace a partir de un constructor que toma como parámetro la dirección de una imagen en formato `.bmp` y dividirá el valor de cada canal y cada pixel por 256 con el fin de limitar el rango de valores en $[0,1]$. Los valores `width` y `height` son extraídos de la cabecera. El constructor admite un parámetro opcional `colorSpace`, que determina si es necesario convertir la imagen cargada de espacio de color. Por el momento solo se soportan dos espacios de color, el espacio Lineal y el espacio sRGB, que son los más comunes.

3.2.1. Mapeo

Puesto que cada textura puede tener distintas resoluciones, es bastante útil definir un sistema de coordenadas relativas a la altura y anchura de una textura. Este sistema se conoce como sistema de coordenadas `u`, `v`. Ambas coordenadas `u` y `v` son valores de punto flotante comprendidos entre $[0,1]$. `u` indica la coordenada horizontal mientras que `v` indica la coordenada vertical. Estas coordenadas son diferentes a las coordenadas `u`, `v` explicadas en Intersección triángulo - rayo aunque tengan el mismo nombre.

También resulta útil definir dos parámetros de transformación para las texturas. Estos son `Tile` y `Offset`. El primero indica el inverso de la escala de la textura, útil por ejemplo si se busca que una textura se repita cierto número de veces y el segundo son los desplazamientos de esta en los dos ejes. Dicha transformación queda contemplada en Listing 3.2

Listing 3.2: Código para obtener valor de una textura

```
__host__ __device__ Vector3 getValueFromCoordinates(int x, int y) {  
  
    Vector3 pixel;  
  
    // Offset and tiling tranforms  
    x = (int)(xTile * (x + xOffset)) % width;  
    y = (int)(yTile * (y + yOffset)) % height;  
  
    pixel.x = data[(3 * (y * width + x) + 0)];  
    pixel.y = data[(3 * (y * width + x) + 1)];  
    pixel.z = data[(3 * (y * width + x) + 2)];  
  
    return pixel;  
}
```

3.2.2. IBL (Image Based Lighting)

Hasta ahora, la luz que llegaba a la escena desde el fondo es una radiación homogénea determinada por un color. Sin embargo, es posible utilizar imágenes para iluminar la escena.

La iluminación basada en imagen ha sido uno de los elementos más relevantes en las técnicas para el renderizado fotorrealista. Se utiliza ampliamente en la industria cinematográfica debido a la complejidad visual que aporta a una escena 3D porque permite captar la iluminación de entornos reales y posteriormente añadirla en escenas digitales. El hecho de poder trasladar la iluminación a un escenario virtual facilita la composición de modelos tridimensionales en películas y series de televisión, donde es necesario juntar una grabación real con un elemento generado por ordenador.

Esta técnica se basa principalmente en usar una fotografía de 360 grados como fuentes de luz formando una esfera alrededor de la escena. Estas fotografías son conocidas como HDRI (del inglés High Dynamic Range Image). A diferencia de las imágenes tradicionales, las cuales normalmente tienen 8 bits de resolución por canal de color, las imágenes HDRI cuentan con valores de punto flotante. El uso de las imágenes HDRI no se limita a la visualización de estas en pantallas de 8 bits de resolución por color como la mayoría de imágenes, sino que el valor de cada píxel será utilizado para realizar las operaciones pertinentes para iluminar la escena. Un ejemplo de este tipo de iluminación se aprecia en la Figura ??



Figura 3.5: Escena iluminada con una imagen HDRI de estudio, nótese como la luz superior incide directamente en la esfera acorde a la dirección

La implementación de esta técnica en el motor de render viene dada por el uso de una imagen en formato .hdr (imágenes en punto flotante). Cada píxel de esta imagen se interpreta como una pequeña fuente de luz direccional en el infinito, orientada hacia el centro de la escena. Así pues, los rayos que no interseccionan en la escena con ningún elemento se considera que interseccionan en el infinito con el HDRI. Por esta razón, al detectar que un rayo sale de la escena, se obtiene la dirección de este, y esta dirección se traduce en las coordenadas polares de la imagen HDRI. Una vez se tienen las coordenadas polares, es posible obtener el valor lumínico del píxel al que apunta dicho rayo. Este será el valor lumínico que se utilice para dicho camino.

Debido a la naturaleza esférica de los mapas de entorno, resulta útil añadir dos funciones que transforman coordenadas esféricas en coordenadas u, v . Estas dos funciones son `sphericalMapping` definida en Listing 3.3 y su inversa `reverseSphericalMapping` definida en Listing 3.4. La primera devuelve las coordenadas u, v para un punto situado en la superficie de una esfera de radio arbitrario mientras que la segunda calcula la posición de un punto en la superficie de una esfera de radio unitario, dadas dos coordenadas u, v .

Listing 3.3: Código para calcular las coordenadas u, v a través de un vector en una esfera de radio determinado

```
__host__ __device__ static inline void sphericalMapping(Vector3 origin,
    Vector3 point, float radius, float& u, float& v) {

    // Point is normalized to radius 1 sphere
    Vector3 p = (point - origin) / radius;

    float theta = acos(-p.y);
    float phi = atan2(-p.z, p.x) + PI;

    u = phi / (2 * PI);
    v = theta / PI;

    limitUV(u,v);
}
```

Listing 3.4: Función inversa al mapeo esférico

```
__host__ __device__ static inline Vector3 reverseSphericalMapping(float
    u, float v) {

    float phi = u * 2 * PI;
    float theta = v * PI;

    float px = cos(phi - PI);
    float py = -cos(theta);
    float pz = -sin(phi - PI);

    float a = sqrt(1 - py * py);

    return Vector3(a * px, py, a * pz);
}
```

3.2.3. Filtrado

Las texturas cuentan con valores discretos y resoluciones limitadas. Esto provoca que la imagen se pixele cuando se muestra cercana a la cámara. Una solución adoptada de manera general en muchos ámbitos es la interpolación de los píxeles vecinos. Actualmente, muchos visualizadores de imágenes no muestran los píxeles naturales, sino una versión modificada de estos. Esto se conoce como filtrado. En la implementación se ha usado un filtrado lineal conocido como interpolación bilineal. Este tipo de filtrado es sencillo de entender e implementar. El valor del punto P comprendido entre los cuatro píxeles más cercanos es la suma ponderada de la distancia del punto a cada píxel en cada dimensión.

```
__host__ __device__ Vector3 getValueBilinear(float u, float v) {

    float x = u * width;
    float y = v * height;

    float t1x = floor(x);
    float t1y = floor(y);

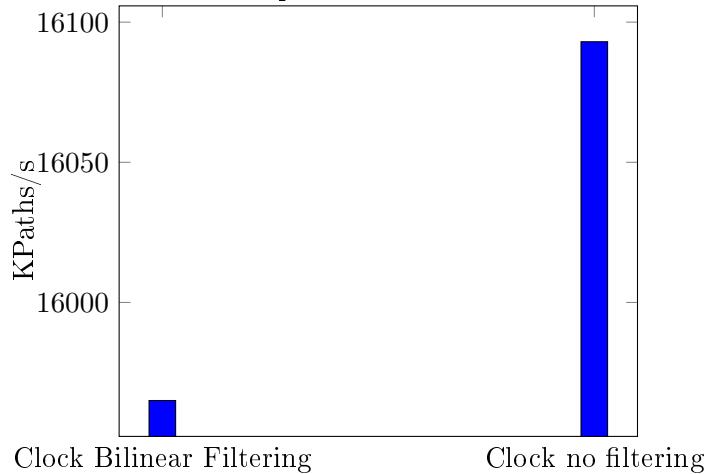
    float t2x = t1x + 1;
    float t2y = t1y + 1;

    // Weights per dimension
    float a = (x - t1x) / (t2x - t1x);
    float b = (y - t1y) / (t2y - t1y);

    // Rounded neighbour values
    Vector3 v1 = getValueFromCoordinates(t1x, t1y);
    Vector3 v2 = getValueFromCoordinates(t2x, t1y);
    Vector3 v3 = getValueFromCoordinates(t1x, t2y);
    Vector3 v4 = getValueFromCoordinates(t2x, t2y);

    // Linear interpolation
    return lerp(lerp(v1, v2, a), lerp(v3, v4, a), b);
}
```

Este método introduce cuatro lecturas del valor de la textura en vez de la única lectura sin usar técnicas de filtrado de manera que resulta conveniente analizar el posible impacto en la eficiencia. Para comprobar esto se ha renderizado la misma escena con y sin filtrado:



El render sin filtrado es un 0.8 % más rápido, una penalización poco sustancial, de manera que se utilizará el filtrado bilineal de manera predeterminada. Se esperaría que esta penalización fuera mayor debido a la multiplicación de accesos a memoria, pero como se explica en el Capítulo 5, la mayor carga de trabajo reside en las intersecciones con los objetos en la escena.

3.2.4. Mapas de normales

A nivel artístico resulta muy útil definir la normal para cada intersección en un triángulo de manera arbitraria, aporta control sobre la dirección en la que luz incide en la superficie además de que permite dar mayor complejidad y detalle a las geometrías contar con la penalización que implica hacer uso de triángulos adicionales Figura 3.6.



Figura 3.6: Esfera con mapa de normales

La mayor parte de la información acerca de los mapas de normales se ha obtenido en LearnOpenGL.com [3], donde se explican las nociones teóricas sobre estos.

Estas normales se suministran a través de texturas de una forma similar a la mencionada anteriormente. La textura que carga la información del mapa de normales cuenta con 3 canales de color. Cada canal se utiliza para definir la coordenada de la normal, siendo el canal rojo la coordenada x, el canal azul la coordenada y y el canal verde la coordenada z. Así pues, la conversión de un color a una normal se hace con el siguiente código: `Vector3 localNormal = (ncolor * 2) - 1;` ya que las coordenadas de la normal se encuentran en el rango [-1, 1] y los valores de color se encuentran en el rango [0,1].

Hasta ahora en ningún momento se ha tenido en cuenta el espacio en el que se encuentran estas normales. Los vectores suministrados por los mapas de normales son locales en el espacio tangencial. El espacio tangencial es un espacio formado por la normal cal-

culada a partir del triángulo, la tangente y la bitangente, tres vectores aproximadamente ortogonales.

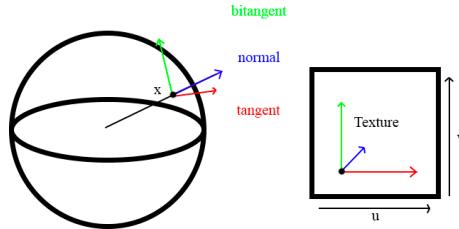


Figura 3.7: Espacio tangencial y su proyección en una textura.

Las normales que se han utilizado hasta ahora para los cálculos de iluminación son normales globales (world normals), en la base ortonormal. La conversión de un vector en el espacio tangencial al global se realiza de la siguiente manera: `worldNormal = (localNormal.x * tangent - localNormal.y * bitangent + localNormal.z * normal).normalized()`

El cálculo de la tangente y bitangente a partir de la normal no es un cálculo trivial, hay infinitos vectores ortogonales que pueden formar esta base. Morten Mikkelsen [9] propone un método estandarizado para calcular este espacio tangencial y además aporta una implementación en C en su repositorio [8]. Gran parte de los programas de diseño 3D utilizan este método o uno compatible. Por esa razón se ha decidido hacer uso de ello. El método en sí se encarga de cada caso específico, como por ejemplo polígonos degenerados, pero en su esencia calcula la tangente en dirección al incremento de la coordenada u en el triángulo y la bitangente en dirección de la coordenada v , como se explica en el artículo de LearnOpenGL [3]

Para hacer uso de la implementación MikkTSpace tan solo hay que definir un archivo adicional que enlace las funciones de callback, las cuales solicitarán parámetros para el cálculo de las tangentes tales como las posiciones de los vértices, el número de triángulos etc.

3.3. Smooth Shading

En Intersección triángulo - rayo se explica cómo las normales son calculadas a partir de la superficie que forma el triángulo. Para modelos con poca cantidad de triángulos, este método de cálculo de la normal de la superficie puede resultar insuficiente.

Un arreglo sencillo consiste en aplicar el método conocido como Smooth Shading.

Para este método es necesario precalcular una normal por cada vértice, algo que hacen casi todos los programas de diseño 3D. En el caso de los ficheros .obj, estas son definidas con el prefijo "vn". Una vez se cuenta con dichas normales, es necesario interpolarlas.

Por ello, en la función `hit()` de `Tri` se añade lo siguiente:

```
#if SMOOTH_SHADING

Vector3 shadingNormal = normals[0] + (normals[1] - normals[0]) * u +
    (normals[2] - normals[0]) * v;
Vector3 shadingTangent = tangents[0] + (tangents[1] - tangents[0]) * u +
    (tangents[2] - tangents[0]) * v;
```

Así pues tanto la normal y la tangente son interpoladas Figura 3.8 a partir de las coordenadas baricéntricas del triángulo u, v , las cuales indican la distancia a cada vértice.

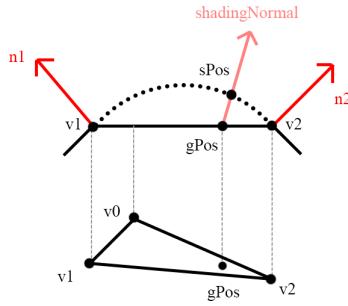


Figura 3.8: Interpolación de normales

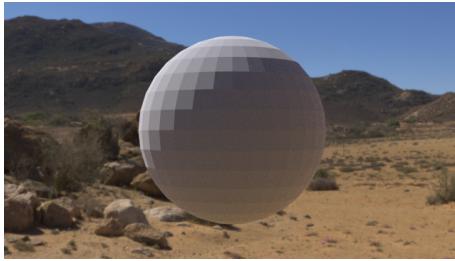


Figura 3.9: Smooth Shading desactivado

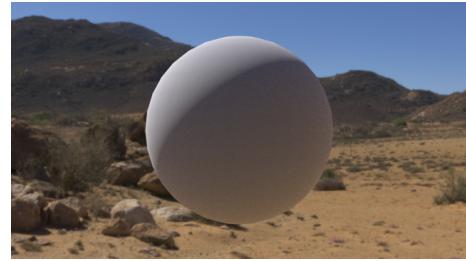


Figura 3.10: Smooth Shading activado

Al implementar esta mejora pueden aparecer artefactos conocidos como "Shadow Terminator Artifact". El uso de una normal interpolada implica que se está haciendo uso de una normal que no coincide con la posición de la superficie. Esta discrepancia crea artefactos visuales presentes hasta en las implementaciones más actualizadas de Blender Cycles Figura 3.11a. Mauricio Vives en [16] propone una solución a este problema y es computar una nueva posición de la superficie a partir de la proyección de la antigua posición al plano que define cada vértice y su normal. Estas tres proyecciones son interpoladas de la misma manera que con las normales, haciendo uso de las coordenadas baricéntricas.

Listing 3.5 muestra el código adaptado a Eleven Renderer.

Listing 3.5: Código Shadow Terminator

```

Vector3 p0 = projectOnPlane(geomPosition, vertices[0], normals[0]);
Vector3 p1 = projectOnPlane(geomPosition, vertices[1], normals[1]);
Vector3 p2 = projectOnPlane(geomPosition, vertices[2], normals[2]);

Vector3 shadingPosition = p0 + (p1 - p0) * u + (p2 - p0) * v;

bool convex = Vector3::dot(shadingPosition - geomPosition,
    shadingNormal) > 0.0f;

hit.position = convex ? shadingPosition : geomPosition;

```

El resultado de la implementación en Eleven Renderer se muestra en la Figura 3.11b



(a) Escena de esfera en Cycles



(b) Escena de esfera en Eleven Renderer

Figura 3.11: Artefactos visuales y solución debidos a Shadow Termination.

3.4. Sombreado BRDF

El sombreado (shading) es el proceso por el cual se asigna un valor de pérdida de energía para un rayo que intersecciona con un punto. Este proceso simula el proceso natural de la interacción entre un haz de fotones y una superficie, dónde parte de los fotones son absorbidos dependiendo de distintos factores como la longitud de onda de los fotones, el tipo de superficie, el tipo de material (los metales reflejarán de manera menos aleatoria que los materiales dieléctricos). La simulación de estas interacciones y simplificación en una función es una ciencia conocida como PBR, o Physically Based Rendering, donde se trata de imitar de manera fiel la realidad física.

En la ecuación de renderizado se encuentra un término denominado BRDF $f_r(\mathbf{x}, \omega_i, \omega_o)$. Hasta ahora ha sido ignorado y simplificado como una superficie Lambertiana perfectamente difusa, es decir que se ha tratado la luz independientemente del ángulo de incidencia ω_i y del ángulo de reflexión del rayo ω_o . En el momento en el que se tienen en consideración estos dos ángulos, se pueden configurar funciones complejas que determinen distintos coeficientes para distintos ángulos. Una función BRDF basada en comportamientos físicos reales como los mencionados anteriormente ofrecerá un resultado acercado a la realidad.

Disney propone una función BRDF conocida como Disney Principled BRDF[1]. Este modelo ha sido ideado por Disney con el fin de adaptar los parámetros del modelo de sombreado a unos parámetros amigables para los artistas. Con esto se pretende dejar de lado los antiguos modelos de sombreado poco intuitivos. Esta función de sombreado es utilizada también por Blender Cycles como modelo predeterminado.

El término "material" y "BRDF" están fuertemente ligados. Un material es un conjunto de valores para estos atributos.

Así pues se procede a explicar estos atributos sintetizados:

- Albedo: Color base.
- Emission: Energía lumínica añadida, el término $L_e(\mathbf{x}, \omega_o)$ de la ecuación de renderizado.
- Roughness: Rugosidad de una superficie.
- Metallic: Como de metálica es una superficie, cuanto menor sea la cantidad, más dieléctrico el material.
- Clearcoat: Capa de barniz superior.
- ClearcoatGloss: Lo pulida que está dicha capa de barniz.

- Anisotropic: Distorsión de la luz propia de los metales.
- Specular: Cantidad de reflejo especular.
- SpecularTint: Color de dicho reflejo.
- Sheen: Cantidad de luz en los bordes del material.
- SheenTint: Color de la capa de luz anterior.
- Subsurface: Sub Surface Scattering falso simulado.

En Eleven Renderer se ha hecho uso de una implementación ya existente [6] de las ecuaciones del shader de Disney adaptada sin utilizar la transmisión. El propio Disney tiene en Github implementaciones de este shader disponibles [14].

Capítulo 4

Optimizaciones estructurales

Un algoritmo con buenos resultados visuales incluye unos costes computacionales asociados. A parte de buscar fotorrealismo y fidelidad gráfica, un motor de renderizado gráfico requiere de mejoras y optimizaciones para alcanzar tiempos competentes.

La importancia de las optimizaciones implementadas en este capítulo no solo se limita a mejorar la productividad de los artistas con mejoras de rendimiento, si no que hace posible la visualización real de escenas que tardarían tiempos del orden de miles de años, reduciendo el enfoque de fuerza bruta a uno más sofisticado.

4.1. Muestreo por importancia

El simple hecho de simular los caminos de los fotones no es eficiente. Se puede buscar una manera para que cada camino cargue con información "más útil". Este procedimiento se denomina "Muestreo por Importancia" y es actualmente una línea de investigación en constante desarrollo ya que como se verá a continuación, permite mejoras muy significativas en cuanto al tiempo de convergencia de la imagen final.

Gran parte de la literatura al respecto se resume en las notas de Importance Sampling for Production Rendering[2] donde se repasan todos los fundamentos teóricos del muestreo por importancia. Así pues en este trabajo solo se va a ofrecer una visión generalizada suficiente para entender la implementación realizada en Eleven Renderer.

Esta implementación consta de tres muestreos por importancia distintos: Muestreo por importancia de BRDF, Estimación de Evento Próximo (NEE) y Muestreo por importancia del mapa de entorno.

Algo que todas las técnicas de muestreo por importancia tienen en común es su funcionamiento primario: Emitir más rayos en las direcciones que más interesan y a su vez, dividir el resultado por la función de distribución de probabilidad (`pdf` en adelante). Esto quiere decir que si se emiten el doble de rayos en una dirección que en otra, será necesario dividir este resultado por dos, puesto que se ha recabado el doble de radiación lumínica. Así pues, también se tendrá que multiplicar el resultado menos muestreado por el doble, ya que al muestrear la mitad, se obtiene la mitad de radiación.

Cuando se ha explicado la aproximación de la Ecuación de Renderizado se ha mostrado un denominador $p(\omega_k)$ que hasta el momento no ha jugado ningún papel. Este denominador es en efecto la función `pdf`.

En la implementación, todo muestreo consiste en una función que genera rayos aleatorios a aquellos lugares donde interesa más muestrear `sample()` y la función de distribución de probabilidad apropiada `pdf()` derivada de la función de muestreo, que ha de devolver la probabilidad con la que se ha elegido ese camino. Como se verá más adelante, no siempre

es fácil derivar esta función de probabilidad cuando se cuenta con una función de muestreo compleja.

4.1.1. Muestreo por importancia de Estimación de Evento Próximo (NEE)

El renderizado por luz directa es un método por el cual solo se tiene en cuenta la primera interacción del rayo con la escena y se comprueba si dicho punto es visible a los elementos lumínicos. Este método converge muy rápido puesto que solo necesita computar una interacción y el fotón recibe información directamente de la luz sin tener que desperdiciar fotones que acaben en zonas oscuras, pero carece del fotorrealismo que ofrece la iluminación global o luz indirecta, puesto no tiene en cuenta la luz reflejada en otros elementos no emisivos.

El muestreo por importancia de Estimación de Evento Próximo trata de juntar estos dos métodos: La iluminación directa y la indirecta. De esta manera, la parte expuesta directamente a elementos lumínicos convergerá rápidamente, además de las partes afectadas por la iluminación global, que lo harán también. Un añadido de este método es la posibilidad de computar la iluminación dada por luces de punto.

Las luces de punto consisten en un punto en el espacio infinitamente pequeño que emite luz de manera uniforme hacia todos los lados. Sin la implementación de la estimación de evento próximo, nunca se recibiría información lumínica de estas luces ya que al ser infinitamente pequeñas no serían alcanzadas por los caminos. No solo eso, NEE también permite una mejor convergencia en las luces de menor tamaño, puesto que originalmente, las luces tienen menos probabilidades de ser alcanzadas cuanto menor sea su área.

En el caso de Eleven Renderer, una nueva función `pointLight` es utilizada. Esta función calcula la luz directa de los puntos de luz. De forma resumida, se elige un punto de luz aleatorio, se toma como radiación inicial el atributo `radiance` para dicho punto de luz y se atenúa cuadráticamente con la distancia entre la luz y el punto a sombrear. Este procedimiento es realmente intuitivo si se trata la luz como una esfera y la radiación emitida como el área de dicha esfera. Finalmente se aplica la función BRDF para la dirección a la luz y el coseno con la normal.

La función `pdf` se obtiene a partir del número de luces. Por otro lado, hay que hacer un cambio de dominio, ya que hasta el momento la función `pdf` solo tiene en cuenta la luz de las luces y no toda la luz de la escena. Este cambio de dominio se hace dividiendo entre el área de media esfera $2.0 * \pi$, ya que es el área del que se mide la radiación entrante.

```

__device__ Vector3 pointLight(Ray ray, HitData hitdata, dev_Scene*
scene, Vector3 point, float& pdf, float r1, Vector3& newDir) {

    PointLight light = scene->pointLights[(int)(scene->pointLightCount *
r1)];

    pdf = scene->pointLightCount / (2.0 * PI);

    newDir = (light.position - point).normalized();

    float dist = (light.position - point).length();

    Vector3 pointLightValue = (light.radiance / (dist * dist));

    Vector3 brdfDisney = DisneyEval(ray, hitdata, newDir);

    return pointLightValue * brdfDisney * abs(Vector3::dot(newDir,
hitdata.normal)) / pdf;
}

```

4.1.2. Muestreo por importancia de entorno

Anteriormente se ha descrito cómo la iluminación HDRI permite resultados visualmente complejos de manera sencilla, simplemente con una imagen de alta profundidad de color. Este método, por otra parte, también cuenta con inconvenientes.

Antes de aplicar cualquier tipo de optimización a este método de iluminación, se estaba trazando un rayo de manera ingenua, recibiendo el color de la imagen a partir de las coordenadas esféricas asumiendo una esfera de radio infinito. Para imágenes homogéneas con luminosidad similar en cada píxel. Este método funciona muy bien, pero la mayoría de estas imágenes HDRI consisten en un paisaje con componentes lumínicos condensados como el sol, que actúa como principal fuente de luz.

Esta situación plantea un problema, y es que los focos de luz como puede ser el sol, concentran un gran porcentaje de luminosidad, mientras que el resto de la imagen no. Esto implica que pocas veces se va a recibir información del sol, lo que va a resultar en "fireflies" (píxeles que cargan una gran radiación lumínica, por la naturaleza aleatoria de las muestras) Figura 4.1a. La forma de enfrentar este problema es igual que el resto de muestreo por importancia: Trazar más rayos a los píxeles más luminosos, ya que serán aquellos que más información aporten a la escena. En la implementación se activa y desactiva con la definición `#define HDRIIS = true/false`



(a) HDRIIS = false



(b) HDRIIS = true

La implementación de este tipo de muestreo por importancia no es trivial. Las imágenes

HDRI utilizadas normalmente en la industria cinematográfica suelen contar con resoluciones extremas ($\tilde{8}K$) y son del orden de millones de píxeles. La búsqueda de los píxeles más brillantes y la asignación de su probabilidad de ser elegidos requerirá de un preprocesamiento previo que facilite dicha búsqueda.

El planteamiento general para este método consiste en precomputar un array del tamaño del número total de píxeles de la imagen HDRI, en el que cada elemento cuente con la iluminación acumulada normalizada hasta el momento de cada píxel Figura 4.2. La función que determina la iluminación de un píxel viene dada por la respuesta logarítmica media de los bastones y conos de los ojos para cada longitud de onda, pero para simplificar se utilizará la suma de cada componente de color. Este array, a efectos prácticos contiene la función de distribución acumulativa discreta normalizada (CDF) y, como está ordenado, es posible hacer búsquedas binarias, las cuales cuentan con una complejidad logarítmica (aproximadamente 25 iteraciones de búsqueda para una imagen de 8K de resolución).

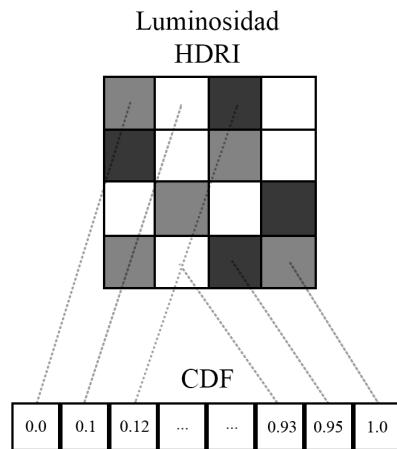


Figura 4.2: Función de distribución acumulada

```

__host__ __device__ inline void generateCDF() {

    int c = 0;
    radianceSum = 0;
    cdf[0] = 0;

    // Total radiance of the HDRI
    for (int j = 0; j < texture.height; j++) {
        for (int i = 0; i < texture.width; i++) {
            Vector3 data = texture.getValueFromCoordinates(i, j);
            radianceSum += data.x + data.y + data.z;
        }
    }

    // CDF calculation
    for (int j = 0; j < texture.height; j++) {
        for (int i = 0; i < texture.width; i++) {
            Vector3 data = texture.getValueFromCoordinates(i, j);
            cdf[c + 1] = cdf[c] + (data.x + data.y + data.z) /
                radianceSum;
            c++;
        }
    }
}

```

La función de cálculo de CDF implementada realiza dos simplificaciones en comparación al método propuesto en la literatura. En primer lugar simplifica la dimensionalidad del array CDF en una sola dimensión (en la literatura se utiliza una segunda dimensión que acumula un sumatorio de radiancia con el fin de acelerar la búsqueda). En segundo lugar, realiza el muestreo por importancia utilizando las coordenadas de las texturas en píxeles y no en coordenadas esféricas (normalmente se muestrea a través de las coordenadas esféricas con el fin de evitar deformaciones en los polos). Se ha determinado que este método modificado es más intuitivo para una versión preliminar, aunque queda sujeto a cambios en posteriores versiones.

Si se quiere obtener un píxel de manera aleatoria proporcional a su iluminación, bastará con buscar en el CDF el intervalo que comprende el valor aleatorio dado. Esta operación de búsqueda también tiene coste logarítmico, ya que es una versión ligeramente modificada de la búsqueda binaria corriente. Este tipo de búsqueda está disponible en la librería estándar de C++ `std::lower_bound`, pero se ha realizado la implementación manual para CUDA. Esta acción es realizada por la función `sample`, que a partir de un número aleatorio, devolverá un píxel del HDRI, siendo los más brillantes los más comunes.

```

// Lower_bound kinda function.
__host__ __device__ int binarySearch(float* arr, float value, int
length) {

    int from = 0;
    int to = length - 1;

    while (to - from > 0) {
        int m = from + (to - from) / 2;
        if (value == arr[m]) return m;
        if (value < arr[m]) to = m - 1;
        if (value > arr[m]) from = m + 1;
    }
    return to;
}

__host__ __device__ Vector3 HDRI::sample(float r1) {
    // Returns position in the array
    int count = binarySearch(cdf, r1, texture.width * texture.height);

    int x = count % texture.width;
    int y = count / texture.width;

    return Vector3(x, y, 0);
}

__host__ __device__ float HDRI::pdf(int x, int y) {

    Vector3 dv = texture.getValueFromCoordinates(x, y);
    float theta = (((float)y / (float)texture.height)) * PI;

    // Semisphere area
    return ((dv.x + dv.y + dv.z) / radienceSum) * texture.width *
    texture.height / (2.0 * PI * sin(theta));
}

```

Por último, como se está muestreando en mayor proporción los píxeles más brillantes, es necesario ofrecer una función `pdf` apropiada. La intuición sugiere que la posibilidad de elegir un píxel entre el resto es la iluminación de dicho píxel dividida entre la suma de toda la iluminación del HDRI (`dv.x + dv.y + dv.z`) / `radienceSum`. No obstante aún faltan matriciales. Puesto que se está calculando la luz de un solo píxel en vez de toda la imagen, es necesario multiplicar el `pdf` por el número total de píxeles de la imagen `texture.width * texture.height`. Además, se aplica el cambio de dominio explicado en Subsección 4.1.1. El cálculo de esta función `pdf` es una extensión del cálculo en NEE, ya que se está considerando un HDRI como un conjunto de luces.

Debido a la deformación de una imagen plana al ser transladada a una superficie esférica, se va a observar un mayor número de muestras en los polos. Este mayor número de muestras en los polos se podría reducir en la función de CDF (es una de las simplificaciones realizadas), pero en vez de eso se añade un término adicional `sin(theta)` que reduce la intensidad de los píxeles más cercanos a los polos.

4.1.3. Muestreo por importancia de BRDF

La función BRDF aporta distintas intensidades lumínicas dependiendo de la dirección entrante y la dirección saliente, por lo que sería mucho más óptimo lanzar más rayos a los sitios en los que esta función sea mayor e ignorar aquellos sitios donde no se aporte mucha información al resultado final.

Un ejemplo que ayuda a visualizar este caso son los materiales perfectamente reflectantes (espejos). Los materiales reflectantes tienen un valor $\text{BRDF} = 1$ para todo rayo entrante simétrico al saliente con la normal como eje de reflexión. Para el resto de direcciones el valor será 0. Por esta razón no tiene sentido emitir rayos aleatorios donde se conozca que el resultado del BRDF será 0.

Así pues, la función `sample` deberá emitir más muestras allá donde la función `brdf` sea mayor. La implementación utilizada contiene una función `DisneySample` que se encarga de esto. Un análisis más detallado de la derivación de esta función a partir de las ecuaciones del shader de Disney se puede encontrar en la implementación de este en el motor de renderizado Autodesk Arnold [12]

A partir de esta función de muestreo, el autor de la implementación ofrece también la función `pdf`. Cabe decir que ambas funciones han sido modificadas y adaptadas para la implementación de Eleven Renderer, eliminando por ejemplo los términos de transmisión en ambas funciones.

Gracias a este muestreo, es posible renderizar materiales metálicos. Anteriormente, la convergencia de estos materiales era lenta e impráctica Figura 4.3a incluso para un número de muestras muy elevado. Tras aplicar el muestreo por importancia el resultado de los materiales metálicos es mucho más nítido Figura 4.3b.



(a) Muestreo uniforme



(b) Muestreo por importancia

Figura 4.3: Diferencias entre distintos muestreos

4.1.4. Muestreo por importancia múltiple

El muestreo por importancia es una herramienta que mejora considerablemente los tiempos de convergencia en determinados escenarios. Como se ha visto hay distintos tipos de muestreos, con distintos puntos fuertes en diversos escenarios. La verdadera utilidad de estos es elegir el muestreo correcto para cada situación. En 1975 se propone una técnica conocida como MIS (Multiple Importance Sampling) [15]. Esta técnica evalúa todas las funciones de muestreo para cada intersección y da mayor importancia con un escalar a aquellas funciones que proporcionen mayor información de la escena.

El cálculo de este escalar se realiza a través de una heurística y debe tener una propiedad esencial: $\sum w(M_i) = 1$. En primer lugar el enfoque más ingenuo es utilizar escalares constantes que sumen 1. Esto garantizará el uso de todos los métodos de muestreo de

manera igualitaria pero con el inconveniente de no aportar ningún tipo de ventaja a los métodos más relevantes.

Por otro lado, una heurística más acertada e intuitiva es ponderar cada método a partir de su **pdf**. Esto quiere decir que será más eficiente dar más importancia al método con mayor **pdf**. Con dos tipos de muestreo diferentes M_1, M_2 quedaría así: $w(M_1) = \frac{pdf_1}{pdf_1 + pdf_2}$, $w(M_2) = \frac{pdf_2}{pdf_1 + pdf_2}$. Se puede comprobar algebraicamente que la suma de ambos pesos es uno.

En los motores de producción se utiliza por lo general una heurística llamada Power Heuristic debido a que Veach [15] la determina empíricamente mejor. Para Eleven Renderer se ha utilizado heurística de ponderación por **pdf** con el fin de ser más comprensivo. En futuras versiones más orientadas a producción se procederá a cambiar esta heurística.

4.2. Estructuras de aceleración

Pese a que este punto cuenta con un nombre genérico, las estructuras de aceleración en el ámbito del renderizado por trazado de rayos hacen referencia a la interacción específica entre geometrías tridimensionales y los rayos generados por la cámara. La intersección Rayo-Triángulo no es en sí una operación demandante a nivel computacional, pero la mayoría de las escenas suelen contar con complejas geometrías del orden de miles de millones de triángulos.

El enfoque más ingenuo y utilizado hasta el momento en este motor de renderizado consiste en evaluar triángulo a triángulo si intersecciona con el rayo dado. Así pues, se observa que es posible reducir el número de comprobaciones de intersecciones aplicando una jerarquía espacial a los triángulos, que descarte parte de ellos para cada interacción, reduciendo así considerablemente el número de operaciones.

Existen diversas formas de estructurar estas jerarquías. Nombrando las más conocidas: Octrees, k-d tree y BVH. En esta implementación se ha usado BVH puesto que se considera que tiene muy buenos resultados.

BVH es acrónimo de "Bounding Volume Hierarchy", traducido como jerarquía del volumen delimitador y consiste en un árbol binario de profundidad definida en el que cada nodo cuenta con un el prisma rectangular que delimita un conjunto de triángulos (en la práctica este prisma consiste en dos puntos en el espacio). Cada nodo interior tiene dos hijos y cada hijo, el volumen delimitador del subconjunto disjunto de los triángulos del nodo padre. Los nodos hoja cuentan con el volumen delimitador y con una lista de los triángulos que contienen.

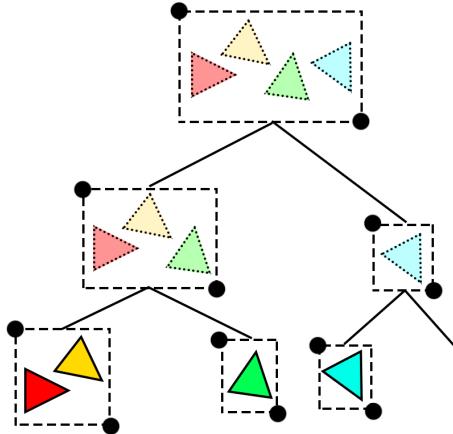


Figura 4.4: División BVH

De esta manera será posible descartar gran parte de los triángulos comprobando los volúmenes por los cuales intersecciónan los rayos.

Esta nueva estructura de datos BVH cuenta con dos operaciones fundamentales: La operación de creación `build` y la operación de recorrido `transverse`. La operación de creación será necesaria solo una vez al principio de cada renderizado de escena y podrá ser realizada en la CPU para mayor simplicidad. La operación de recorrido sin embargo será ejecutada cada vez que se quiera comprobar la interacción Rayo-Triángulo.

El algoritmo empleado es una interpretación del libro PBRT, pero difiere en varios detalles. Pharr et al. [11] almacena un solo triángulo por cada nodo hoja, a diferencia del código implementado en Eleven Renderer, donde hay una profundidad máxima definida como `BVH_DEPTH`, y los nodos hojas almacenan los índices de uno o varios triángulos.

4.2.1. Operación de generación de BVH (CPU)

La operación de generación parte de la generación recursiva de un árbol binario por profundidad, añadiendo información adicional a los nodos.

1. Para cada nodo que contiene un conjunto de triángulos, se calcula el volumen delimitador. Este volumen viene dado por el vértice menor y mayor del volumen `b1`, `b2`.
2. Se separa el conjunto de triángulos en dos subconjuntos disjuntos. El cómo se divide este conjunto se delega a la función `divideSAH()`, que hará uso de la heurística de superficie de área para decidir qué triángulos van a cada subconjunto.
3. Finalmente se aplica recursión para los hijos generados en caso de residir en un nodo interior, o se termina la recursión y se almacenan los índices `from` y `to` de los triángulos que estos contienen en caso de ser un nodo hoja.

Los delimitadores son dos puntos que representan el volumen que contiene una geometría. El cálculo de estos se hace cogiendo las coordenadas mínimas y máximas. La operación de unión de estos delimitadores se hace aplicando el mismo esquema de coordenadas mínimas y máximas.

```

void buildAux(int depth, std::vector<BVHTri>* _tris) {

    Vector3 b1, b2;

    if (depth == 0)
        totalTris = _tris->size();

    if(depth == 7)
        printf("\rAllocated tris: %d / %d, %d%%", allocatedTris,
              totalTris, (100 * allocatedTris) / totalTris);

    bounds(_tris, b1, b2);

    if (depth == DEPTH) {

        nodes[nodeIdx++] = Node(nodeIdx, b1, b2, triIdx, triIdx +
                               _tris->size(), depth);

        for (int i = 0; i < _tris->size(); i++)
            triIndices[triIdx++] = _tris->at(i).index;

        allocatedTris += _tris->size();
    }
    else {

        nodes[nodeIdx++] = Node(nodeIdx, b1, b2, 0, 0, depth);

        std::vector<BVHTri>* trisLeft = new std::vector<BVHTri>();
        std::vector<BVHTri>* trisRight = new std::vector<BVHTri>();

        divideSAH(_tris, trisLeft, trisRight);

        buildAux(depth + 1, trisLeft);
        buildAux(depth + 1, trisRight);

        trisLeft->clear();
        trisRight->clear();

        delete trisLeft;
        delete trisRight;
    }
}

```

Una duda que puede surgir es cómo se almacenan los triángulos de manera eficiente en estos árboles, ya que almacenar tantas listas como nodos hoja hay, es muy ineficiente en términos de memoria. Aquí es donde entran en juego los índices `from` y `to` mencionados anteriormente. Se hace uso de una lista de ordenación de triángulos denominada `triIndices`, la cual contiene índices de los triángulos en la lista original ordenados por nodos. Los nodo hoja tendrán pues dos índices indicando desde qué índice (`from`) hasta qué índice (`to`) es necesario leer de manera inclusiva en esta lista para recuperar los triángulos almacenados por dicho nodo.

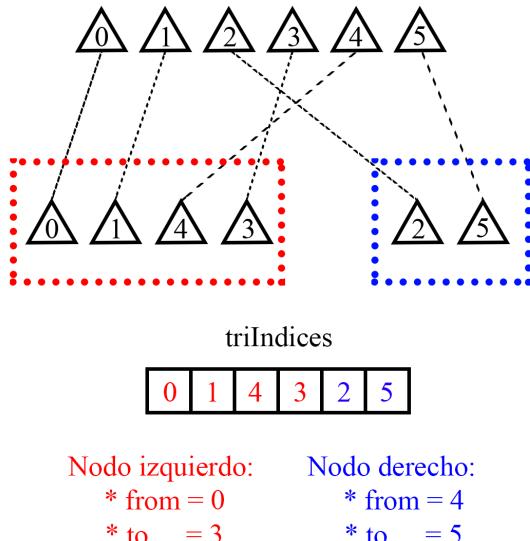


Figura 4.5: triIndices

Una vez definida la función de generación de árboles BVH solo queda decidir cómo se van a particionar los triángulos en cada nodo.

El hecho de como separar los nodos hijos de manera óptima no es una tarea trivial, en esta sección se mostrarán comparativas mostrando como los tiempos de recorrido varían dependiendo del método elegido, además de que algunos métodos tardan más en construirse pero permiten recorridos más rápidos mientras que otros métodos permiten la construcción de árboles en tiempo lineal pero no ofrecen tan buenos resultados a la hora de recorrerlos. Estos últimos son más usados en aplicaciones de tiempo real que requieren construir árboles BVH en milisegundos.

La aplicación de este trabajo requiere de varios segundos e incluso minutos por cada escena renderizada, por lo que existe la libertad de construir árboles con métodos más lentos que no afectarán tan negativamente al tiempo total de renderizado.

Se han implementado dos métodos distintos para comparar su rendimiento. Uno de ellos es un método ingenuo y el segundo es un método utilizado en producción conocido como división por heurística de superficie (SAH).

Partición por plano:

En el libro *Physically based rendering: From theory to implementation* [11] se propone un método de partición sencillo. A continuación se explica una interpretación algo distinta de este método que hace uso de los centroides de los triángulos en vez de utilizar el vértice 0 como hace el libro.

Este método no es práctico y se usa tan solo de forma comparativa. Consiste en elegir la dimensión más grande de la caja que envuelve a los triángulos y partirla por la mitad. Posteriormente se recorrerán todos los triángulos y se dividirán por dicho plano. Para saber si un triángulo se encuentra a un lado o al otro se interpretan los triángulos como un punto en el espacio cuya posición es el centroide.

```

static void dividePlane(std::vector<BVHTri>* tris, std::vector<BVHTri>*
trisLeft, std::vector<BVHTri>* trisRight) {

    Vector3 b1, b2;

    bounds(tris, b1, b2);

    Vector3 l = (b2.x - b1.x, b2.y - b1.y, b2.z - b1.z);

    if (l.x > l.y && l.x > l.z) {

        for (int i = 0; i < tris->size(); i++) {
            if (tris->at(i).tri.centroid().x > b1.x + l.x / 2) {
                trisLeft->push_back(tris->at(i));
            }
            else {
                trisRight->push_back(tris->at(i));
            }
        }

    } else if (l.y > l.x && l.y > l.z) {
        for (int i = 0; i < tris->size(); i++) {
            if (tris->at(i).tri.centroid().y > b1.y + l.y / 2) {
                trisLeft->push_back(tris->at(i));
            }
            else {
                trisRight->push_back(tris->at(i));
            }
        }
    } else {
        for (int i = 0; i < tris->size(); i++) {
            if (tris->at(i).tri.centroid().z > b1.z + l.z / 2) {
                trisLeft->push_back(tris->at(i));
            }
            else {
                trisRight->push_back(tris->at(i));
            }
        }
    }
}

```

Heurística SAH:

4.2.2. Operación de recorrido (GPU)

El recorrido de árboles es una tarea intuitivamente recursiva y cuenta con una implementación conocida y sencilla. No obstante la recursión es un proceso indeseado en CUDA debido a las pilas relativamente pequeñas de los dispositivos de aceleración gráfica. Así pues también generan una divergencia superior, un proceso con alta penalización en el rendimiento.

Para demostrar esto se ha hecho uso de dos implementaciones de la función de recorrido. La primera es una función recursiva de recorrido de árboles binarios:

```

__host__ __device__ void transverseAux(Ray ray, Hit& nearestHit, Node&
node) {

    if (node.depth == BVH_DEPTH) {
        intersectNode(ray, node, nearestHit);
        return;
    }

    Node lChild = leftChild(node.idx, node.depth);
    Node rChild = rightChild(node.idx, node.depth);

    if (intersect(ray, lChild.b1, lChild.b2))
        transverseAux(ray, nearestHit, lChild);

    if (intersect(ray, rChild.b1, rChild.b2))
        transverseAux(ray, nearestHit, rChild);
}

```

Karras explica en el blog de desarrolladores de Nvidia [5] cómo usar las estructuras BVH para detectar colisiones físicas entre objetos en la GPU. Así pues, la segunda implementación viene dada a partir de ciertas modificaciones al código de recorrido propuesto, donde se puede adaptar a un caso más específico como es la intersección de Rayo-BVH necesaria en Eleven Renderer. Esta implementación rompe con la recursividad a partir de una pila de nodos, quedando así una función iterativa.

```

__device__ void transverse(Ray ray, Hit& nearestHit) {

    Node stack[64];

    Node* stackPtr = stack;

    (*stackPtr++).valid = false;

    Node node = nodes[0];

    do {

        Node lChild = leftChild(node.idx, node.depth);
        Node rChild = rightChild(node.idx, node.depth);

        bool lOverlap = intersect(ray, lChild.b1, lChild.b2);
        bool rOverlap = intersect(ray, rChild.b1, rChild.b2);

        if (node.depth == (BVH_DEPTH - 1) && rOverlap)
            intersectNode(ray, rChild, nearestHit);

        if (node.depth == (BVH_DEPTH - 1) && lOverlap)
            intersectNode(ray, lChild, nearestHit);

        bool traverseL = (lOverlap && node.depth != (BVH_DEPTH - 1));
        bool traverseR = (rOverlap && node.depth != (BVH_DEPTH - 1));

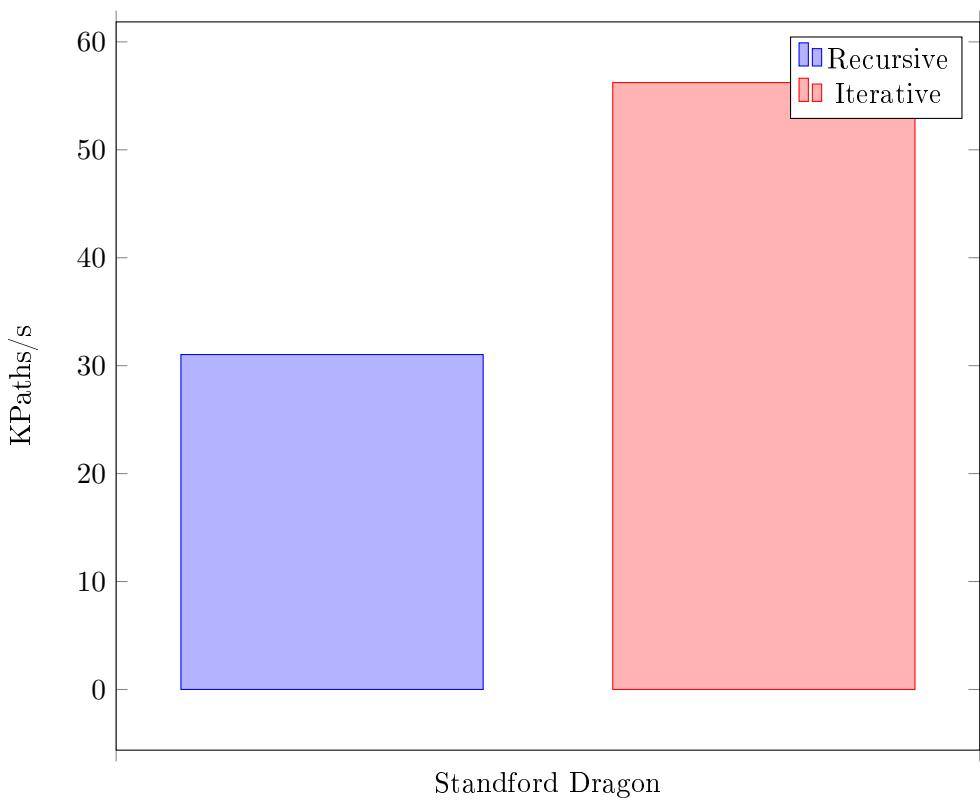
        if (!traverseL && !traverseR) {
            node = *--stackPtr;

        } else {
            node = (traverseL) ? lChild : rChild;
            if (traverseL && traverseR)
                *stackPtr++ = rChild;
        }

    } while (node.valid);
}

```

Una breve evaluación de ambas funciones muestran una clara ventaja en la implementación iterativa. Además ha sido necesario limitar la profundidad a 6 nodos debido a las limitaciones mencionadas anteriormente. El uso de más profundidad en el entorno utilizado provoca el bloqueo del kernel



Comparación de heurísticas:

Para justificar el uso de la heurística por superficie de área se muestra una ejecución de ambas heurísticas con tres modelos ampliamente utilizados en evaluaciones comparativas de computación gráfica:

- Suzanne: 968 triángulos
- Standford Bunny: 4968 triángulos
- Standford Dragon: 871306 triángulos



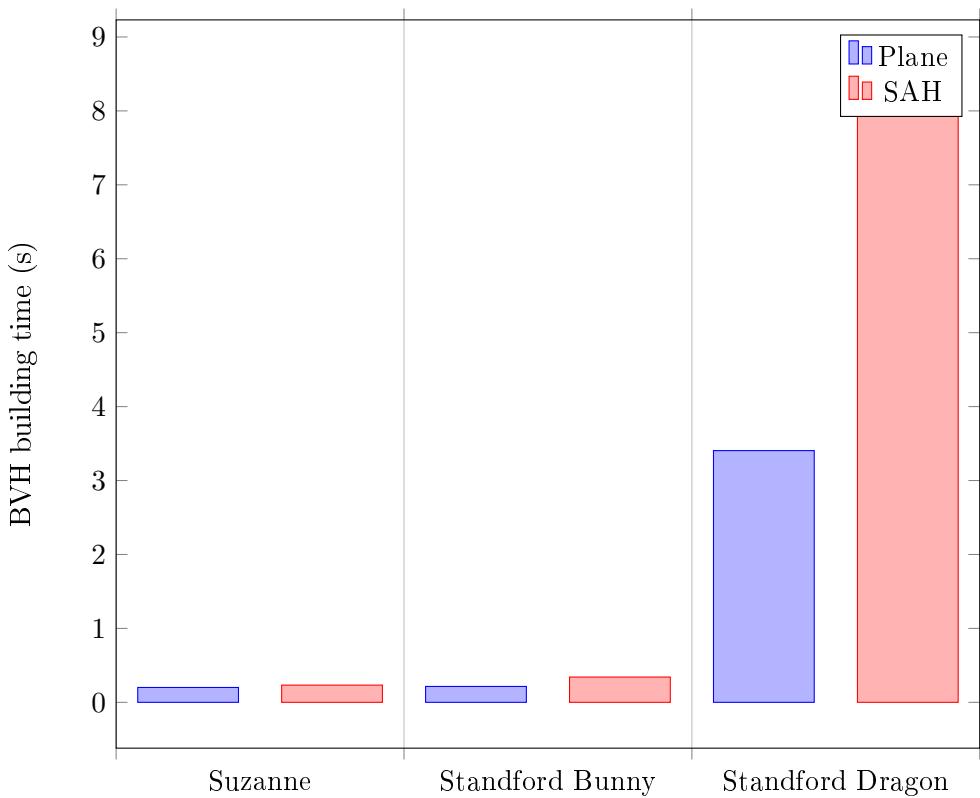
Figura 4.6: Standford Bunny.



Figura 4.7: Standford Dragon.

Esta evaluación ha sido llevada a cabo con los siguientes parámetros:

`BVH_DEPTH = 20, BVH_SAHBINS = 12, MAXBOUNCES = 5, SMOOTH_SHADING = true,`



4.2.3. Evaluación:

Finalmente es necesario elegir los parámetros para `BVH_DEPTH` y `SAH_BINS`. Puesto que existen varios tipos de escenas con distintas geometrías, se va a dejar de lado cualquier tipo de enfoque analítico y se va a hacer uso de medidas en casos reales. Se procede a realizar una evaluación con distintos parámetros.

En el caso de la selección de `SAH_BINS`, Indigo Wald propone 16 como límite [17] debido a la insustancial mejora de rendimiento para valores mayores. Esto se ha podido comprobar directamente en Eleven Renderer donde se ha llevado a cabo la construcción y recorrido de BVH para la escena "Standford Dragon" variando la cantidad de contenedores Figura 4.8. Se comprueba que la velocidad de recorrido se estanca para los valores 12-16, mientras que el tiempo de generación permanece lineal.

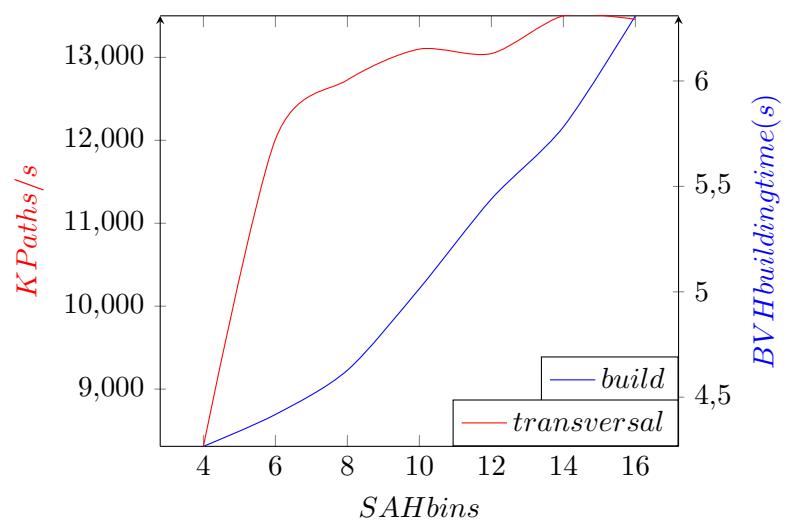


Figura 4.8: Comparación de tiempos de construcción y eficiencia de recorrido para distinto número de contenedores

Capítulo 5

Evaluación

Para llevar a cabo la evaluación del algoritmo se ha hecho uso de la herramienta NVIDIA Nsight Compute. Esta herramienta proporciona un análisis completo sobre la ejecución de un kernel, así como información relevante que puede dar pistas de dónde están los cuellos de botella y qué partes conviene optimizar.

Tras crear un proyecto y acceder al perfil de `renderingKernel`, el primer dato de relevancia que se ha buscado ha sido que parte del código es la más ejecutada. Conociendo este detalle se puede evitar el esfuerzo de optimizar partes poco relevantes. Para conocer este dato es necesario acceder a la pestaña `Source` de la aplicación. En esta pestaña se visualiza el código junto a las instrucciones pptx y a la derecha un mapa de calor que indica en qué partes del código frecuentan más los filtros seleccionados en la pestaña `Navigation`.

Se ha seleccionado el filtro de instrucciones ejecutadas donde se han podido localizar dos principales puntos calientes. El primero de los dos es el acceso a los hijos en los nodos del árbol BVH. El segundo es en las funciones de mínimo y máximo. Esto es de esperar ya que son funciones aritméticas que se utilizan en el cálculo de la intersección del rayo con los nodos del árbol.

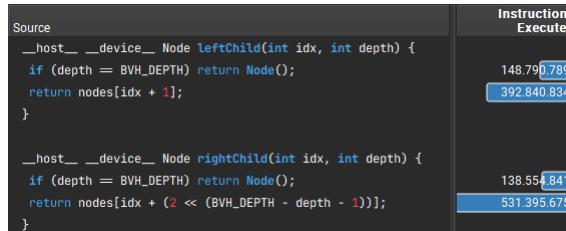


Figura 5.1: Nº instrucciones ejecutadas funciones leftChild, righChild

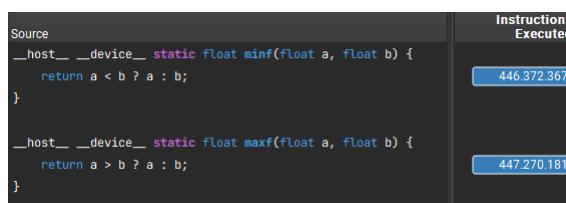


Figura 5.2: Nº instrucciones ejecutadas funciones minf/maxf

Estos resultados muestran que la mayoría de las instrucciones ejecutadas se encuentran en el recorrido de los árboles BVH. No es de extrañar que la mayoría de los esfuerzos de

optimización del trazado de rayos en el estado del arte residan en estructuras de aceleración más óptimas y compactas.

NVIDIA Nsight Compute también ofrece advertencias de que partes pueden resultar problemáticas o subóptimas en una arquitectura de GPU en el panel **Details**. La ejecución de Eleven Renderer provoca el aviso de un error común en arquitecturas paralelas y es el acceso no secuencial de la memoria. Este fenómeno ocurre cuando los hilos de un wrap no acceden a posiciones contiguas, y en el caso de esta evaluación la advertencia redirige a la parte del código que accede a los hijos del árbol BVH. Esto es de esperar puesto que las estructuras de los árboles no ofrecen buena secuencialidad.

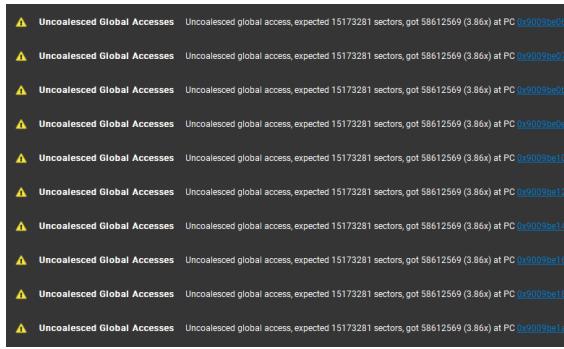


Figura 5.3: Advertencias de acceso no secuencial

5.1. Análisis Roofline

El modelo Roofline es utilizado en el análisis de eficiencia de aplicaciones de altas prestaciones. Es un modelo que simplifica la visión del hardware y software mostrando un posible techo de eficiencia. NVIDIA Nsight Compute ofrece este análisis para poder analizar posibles deficiencias y optimizaciones.

La línea superior es una cota superior de 29.23 TFLOPS

El análisis de ejecución de Eleven Renderer para precisión simple Figura 5.4 ha resultado en 0.36 FLOP/byte de intensidad aritmética y 124.47 GFLOPS de eficiencia, mientras que el techo para una intensidad aritmética de 0.36 FLOP/byte está en 330.65 GFLOPS. Esto indica que el algoritmo está corriendo a un 37.6 % de su capacidad máxima teórica según este modelo siempre y cuando la intensidad aritmética no varíe. Este resultado es razonable, indica que no se está infrautilizando el acelerador gráfico, además indica que el algoritmo tiene una gran dependencia de memoria al encontrarse el punto ubicado muy a la izquierda.

Si se quisiera optimizar más aún este motor, un buen camino sería romper esta dependencia.

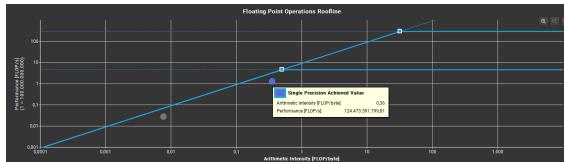


Figura 5.4: Análisis Roofline

Capítulo 6

Port oneAPI

oneAPI es un modelo de programación unificado cuyo fin es la implementación de aplicaciones en distintos tipos de aceleradores.

oneAPI tiene un módulo orientado a la informática gráfica en el que se incluyen herramientas para motores de renderizado. Entre ellas se encuentran Open Image Denoise, una librería que permite renders más detallados con un menor número de muestras gracias al aprendizaje profundo, Embree, que es actualmente el conjunto de kernels más avanzado y eficiente de trazado de rayos gracias a micro optimizaciones adaptadas a distintas arquitecturas, OSPRay, una librería de trazado de rayos y rendering y Open VKL, un kernel de trazado de rayos volumétrico.

Además oneAPI ofrece una plataforma de traducción de código automática que permite convertir código C++ y CUDA a oneAPI. En este capítulo se va a proceder a explicar la conversión realizada de Eleven Renderer, así como los detalles y cambios que han sido necesarios hacer para adaptar esta implementación. También se hará una breve evaluación con el fin de valorar la usabilidad y eficiencia de esta librería y herramienta de portabilidad.

La herramienta en cuestión es DPCT

Capítulo 7

Conclusiones

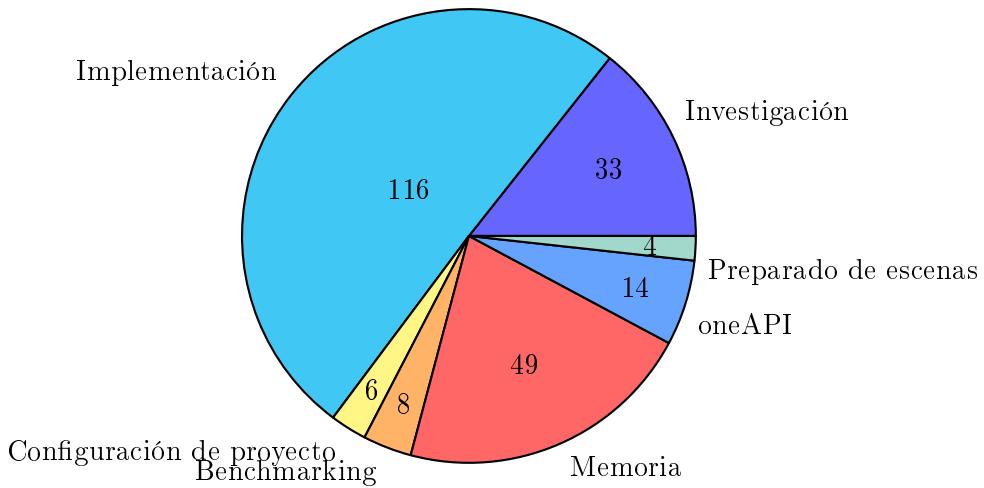
Desarrollar software con el fin de ser ejecutado en aceleradores gráficos puede ser una tarea relativamente sencilla en una primera instancia. El verdadero interés está conseguir el mayor rendimiento y exprimir estas arquitecturas, teniendo en cuenta sus limitaciones y sus puntos fuertes.

Haciendo referencia al desarrollo de un motor de renderizado fotorrealista, han habido grandes desafíos. En primer lugar, el renderizado gráfico cuenta con un gran trasfondo teórico y matemático del que conviene conocer para poder explotar a fondo cualquier optimización. Este trabajo ha expuesto solo la implementación y la evaluación de este software, sin hacer demasiado hincapié en las bases teóricas. Como se mencionaba en la introducción, *Physically based rendering: From theory to implementation* [11] es un excelente recurso para esto, denominado por la comunidad como "La biblia del renderizado PBR".

Por otro lado, la programación de Eleven Renderer sido mucho más tediosa de lo esperado. Mientras que mejoras como la construcción de estructuras de aceleración a primera vista podrían imponer por su complejidad, han resultado ser más sencillas de implementar que mejoras más sutiles y simples. Esto es debido a la falta de opciones de depuración en GPU. Mientras que el desarrollo en CPU cuenta con una infinidad de herramientas para analizar el funcionamiento y la ejecución, CUDA solo ofrece un conjunto muy básico de herramientas de análisis y depuración. Así pues, muchas de las soluciones a problemas que han ido surgiendo han tenido que ser resueltas por prueba y error, un enfoque bastante poco óptimo.

Además el enfoque a lo largo del trabajo de implementar todas las opciones posibles en el tiempo dado tiene la desventaja de no contar con una implementación asentada y limpia, si no una en constante cambio y consecuentemente más complicada de analizar. Aún así este enfoque ha hecho posible abarcar todas las bases de un motor de producción mínimo, y aunque en ciertas secciones carece de trasfondo teórico, da una visión global de todos los componentes necesarios.

Algo que se ha observado tras el desarrollo de Eleven Renderer es la carencia en la industria de estándares definidos. Por un lado, no existe una definición de materiales PBR común. El formato .obj es capaz de incluir archivos .mtl, pero éstos sólo contienen información de modelos de sombreado anticuados. Existen propuestas de extender estos formatos pero en la práctica cada programa utiliza su propio formato de archivo.



7.1. Trabajo Futuro

Eleven Renderer cuenta con las características visuales mínimas de un motor de producción, así pues deja la puerta abierta a un futuro desarrollo comercial o con fines educativos. No obstante, el siguiente desafío si se quiere seguir produciendo es adaptar la implementación a las suites de software 3D a través de plugins que enlacen las escenas con el motor.

Queda también una larga lista de mejoras, entre ellas:

- Mayor variedad de shaders (además del shader de Disney).
- Incluir refracción, puesto que por el momento los materiales transparentes son incompatibles con el motor.
- Implementar eliminadores de ruido como Open Image Denoise o NVIDIA OptiX™ AI-Accelerated Denoiser.
- Añadir un motor de postprocesado más potente con filtros como Bloom o correcciones de color
- Añadir más formatos de luces
- Experimentar con Embree en CUDA

Queda pendiente también realizar a fondo un análisis exhaustivo de eficiencia. En el trabajo se han dado pinceladas en este aspecto, por ejemplo la comparación entre búsquedas recursivas o iterativas en GPU, pero hay un centenar de posibles optimizaciones y de prácticas recomendadas para exprimir la eficiencia de la programación en paralelo.

Conclusions

7.2. Future work

Capítulo 8

Anexo

8.1. Manual

Tras compilar el repositorio o haber descargado un ejecutable, es preciso incluir en la línea de comandos tres parámetros. El primero es el directorio donde se encuentra la escena, el segundo es el número de muestras deseado y el tercero el archivo de salida en formato .bmp de la imagen resultante.

8.1.1. Formato de escenas

Debido a la falta de consenso en cuanto a formatos en la industria, se ha utilizado un formato de escenas intentando respetar los estándares más comunes. Así pues una escena se define como un directorio.

Dentro este directorio debe incluir en su interior 3 carpetas

- Objects: utilizada para las geometrías en formato .obj. Es necesario que estas geometrías hayan sido trianguladas previamente puesto que el parser desarrollado está limitado a triángulos.
- Textures: utilizada para las texturas. Actualmente solo se permiten texturas para los atributos: Albedo, Emission, Roughness, Metallic, Normal. Las texturas tienen que estar en formato bmp de 24 bits. El formato de nombre es el siguiente: nombredelmaterial_tipodemap.bmp. Por ejemplo: material1_albedo.bmp, material1_metallic.bmp. No es necesario que se definan todas las texturas, si no existe alguna se ignorará y se utilizará el valor de color definido en el archivo scene.json. En caso de no existir tampoco ese valor, se utilizará el valor por defecto. Las texturas de albedo y emisión deberán encontrarse en espacio de color sRGB mientras que el resto de texturas deben estar en un espacio lineal. Esto no es respetado por muchos motores de renderizado y es dependiente de la implementación.
- HDRI: utilizada para los mapas de entorno. Dentro albergará los archivos en formato .hdr de los mapas de entorno.

Además será obligatorio incluir un archivo llamado scene.json. Este archivo ha de incluir la información de la escena necesaria. Se muestra un ejemplo como plantilla:

```
{
```

```

'camera' : {'xRes' : 1280, 'yRes' : 720, 'position' : {x : 0, y : 1, z : 2},
            'focalLength' : 0.05, 'focusDistance' : 1, 'aperture' : 2.8},
'materials' : [{name' : 'mat1'}, {'name' : 'mat2', 'albedo' : {'r' : 1, 'g' :
    0, 'b' : 0}, 'roughness' : 0.2}],
'objects' : [{name' : 'obj1', 'material' : 'mat1'}, {'name' : 'obj2',
    'material' : 'mat2'}],
'hdri' : {'name' : 'hdri', 'xOffset' : 0.5},
}

```

El ejemplo mostrado deberá contener dos objetos dentro de la carpeta Objects: obj1.obj y obj2.obj. El material mat1 utilizará las texturas que empiecen por mat1_...bmp mientras que el mat2 utilizará los valores rgb(1,0,0) para albedo y el valor 0.2 para roughness. Para más ejemplos consultar la carpeta Scenes del repositorio.

8.2. Galería

Bibliografía

- [1] Brent Burley and Walt Disney Animation Studios. Physically-based shading at disney. In *ACM SIGGRAPH*, volume 2012, pages 1–7. vol. 2012, 2012.
- [2] Mark Colbert, Simon Premoze, and Guillaume François. Importance sampling for production rendering. In *ACM SIGGRAPH*, 2010.
- [3] Joey de Vries. Learnopengl. <https://learnopengl.com/Advanced-Lighting/Normal-Mapping>.
- [4] James T Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.
- [5] Terro Karras. Thinking parallel, part ii: Tree traversal on the gpu. <https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/>.
- [6] knightcrawler25. GLSL-PathTracer. <https://github.com/knightcrawler25/GLSL-PathTracer/>, 2020.
- [7] Jean-Henri Lambert. *JH Lambert,... Photometria, sive de Mensura et gradibus luminis, colorum et umbrae.* sumptibus viduae E. Klett, 1760.
- [8] Morten Mikkelsen. Mikktospace. <https://github.com/mmikk/MikktSpace>.
- [9] Morten Mikkelsen. Simulation of wrinkled surfaces revisited. 2008.
- [10] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28, 1997.
- [11] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.
- [12] Weng Shih-Chin. Implementing disney principled brdf in arnold. <https://shihchinw.github.io/2015/07/implementing-disney-principled-brdf-in-arnold.html>.
- [13] Peter Shirley. Ray tracing in one weekend. *Amazon Digital Services LLC*, 1, 2016.
- [14] Walt Disney Animation Studios. brdf. <https://github.com/wdas/brdf>.
- [15] Eric Veach and Leonidas J Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 419–428, 1995.
- [16] Mauricio Vives. shading_position.hsls. <https://gist.github.com/pixnblox/5e64b0724c186313bc7b6ce096b08820>.

- [17] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40. IEEE, 2007.