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

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 realionadas 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 mejor 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, siendo RTX (el nombre de las dos últimas series) acrónimo de Ray Tracing Texel eXtreme".

No obstante la implementación de este sistema de renderizado peca de una gran demanda computacional, es por ello 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 las bases teóricas además de explicar la implementación en CUDA de un motor de renderizado fotorrealista con características a la orden del día de otros motores comerciales. Así pues este trabajo también 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; 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 expondrá 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

Este motor gráfico tiene sus raices en un trabajo final previo realizado en la asignatura "Programación de GPUs y aceleradores". En este trabajo previo se hizo una prueba de concepto de un motor "Path tracing"sin ningún tipo de optimización ni mejora visual.

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.

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

Cuadro 1.1 Timeline

Noviembre 2020	Incialización 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

Capítulo 2

Fundamentos del algoritmo 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. Aproximación de la ecuación de renderizado

$$L_{\mathrm{o}}(\mathbf{x},\omega_{\mathrm{o}}) = L_{\mathrm{e}}(\mathbf{x},\omega_{\mathrm{o}}) \; + \int_{\Omega} f_{\mathrm{r}}(\mathbf{x},\omega_{\mathrm{i}},\omega_{\mathrm{o}}) L_{\mathrm{i}}(\mathbf{x},\omega_{\mathrm{i}})(\omega_{\mathrm{i}} \cdot \mathbf{n}) \, \mathrm{d}\,\omega_{\mathrm{i}}$$

La ecuación de renderizado [2] 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.

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}.

El primer término $L_{\mathrm{e}}(\mathbf{x},\omega_{\mathrm{o}})$ 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 $f_{\mathrm{r}}(\mathbf{x},\omega_{\mathrm{i}},\omega_{\mathrm{o}})$ 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 $L_{\mathrm{i}}(\mathbf{x},\omega_{\mathrm{i}})$ hace referencia a toda la energía lumínica entrante de todas las direcciones posibles.

El tercer coeficiente $(\omega_{\mathrm{i}} \cdot \mathbf{n})$ es el producto de la ley del coseno de Lambert [3], un escalar que atenúa los ángulos menos pronunciados con la normal de la superficie.

2.2. Esquema simplificado del trazado de rayos

Habiendo definido la ecuación de renderizado, el siguiente paso es explicar la implementación hecha de los fundamentos del algoritmo de Path Tracing, ya que posteriormente se irá mejorando paso por paso este algoritmo básico.

En su esencia consiste en trazar rayos o çaminos"desde una cámara virtual a una escena tridimensional, simulando así un modelo simplificado de fotones y sus interacciones con la escena. Tras cada interacción con la escena, estos caminos de fotones tienen una pérdida de energía que se acumulará en cada píxel y definirá el color de este, como si del sensor de una cámara real se tratara.

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

Las 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`.

Las geometrías `MeshObject` por otra parte consisten en un conjunto de triángulos `Tri`, los cuales a su vez consisten en 3 puntos tridimensionales `Vector3 vertices[3]`.

Los 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 pues un color consiste en un vector `Vector3(R,G,B)`.

Habiendo definido estos elementos en la escena, se procederá a transferirlos a la GPU, donde se realizará el resto de computaciones más demandantes. Esto es realizado por la función `cudaError_t renderSetup(Scene* scene)`, la cual 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 para los objetos, requiriendo así una copia profunda de los atributos en formato array o punteros, además de un proceso posterior denominado "pointer binding".

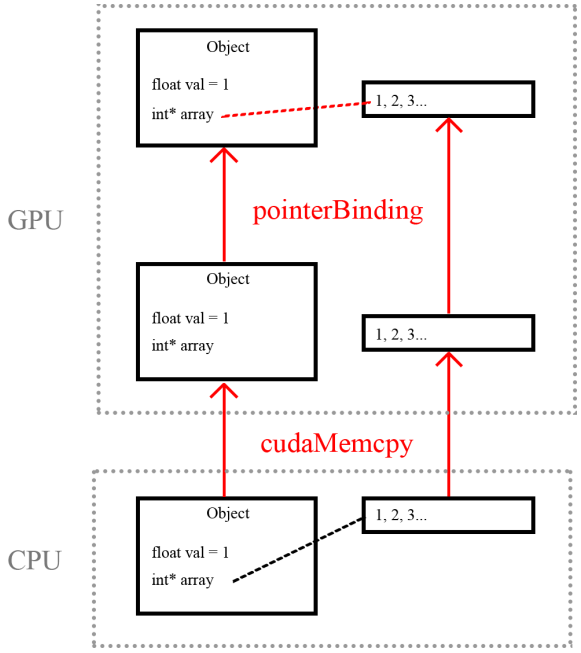


Figura 2.1: Pointer Binding

Una vez están todos los componentes necesarios en la GPU, es preciso llamar a un kernel para configurar el motor en la GPU. Este kernel `setupKernel` se hace cargo de inicializar los bufferes de píxeles y contador de rayos, inicializar `curand` (la librería utilizada para generar números aleatorios en CUDA) y por último asignar a cada objeto el índice de triángulos a l

2.2.1. Trazado de rayo desde la cámara

Para simular el trazado del rayo desde la cámara hasta la escena, se simula de manera simplificada como funcionaría una cámara estenopéica. Se calcula la posición del sensor por la cual se trazará el rayo a partir de las coordenadas `x` e `y`.

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.

```
__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 antialasing
    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);
}
```

2.2.2. 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 intersectan 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 objecto 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 intersectado correctamente.

  Para la implementación se ha hecho uso del algoritmo Fast Minimum Storage Ray/Triangle Intersection[4]. En este paper se explica detalladamente el algoritmo de intsersección, mientras este trabajo se limita a implementar dicho algoritmo a partir de una adaptación de la implementación en C que el autor ofrece.

```
__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;
```

2.3. Renderizado progresivo

Una ventaja de los motores de renderizado más modernos es el renderizado progresivo. Esto implica que las muestras se van acumulando poco a poco a lo largo de la imagen hasta que termina 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. Se ha decidido hacer 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 hará 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 ha implementado con el tipo de datos `cudastream_t` de la API de CUDA. Han sido necesarios dos flujos, uno denominado `kernelStream` y otro denominado `bufferStream`. Los kernels de inicialización y renderizado correrán en el primero, mientras que la función que obtiene el buffer, será lanzada en el segundo.

La función que extrae el buffer de píxeles de la GPU a la CPU es la siguiente:

```
    cudaError_t getBuffer(float* pixelBuffer, int* pathcountBuffer, int size) {

        cudaStreamCreate(&bufferStream);

        cudaError_t cudaStatus = cudaMemcpyFromSymbolAsync(pixelBuffer,
            dev_buffer, 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;
}
```

Hace uso de la función `cudaMemcpyFromSymbolAsync` para realizar la copia asíncrona antes mencionada. También se hace copia del buffer de la suma de rayos emitidos por píxel con el fin de realizar métricas de eficiencia.

Capítulo 3

Mejoras estructurales

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 va a hacer 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 de código que hace el cálculo del desenfoque.

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 es el siguiente:

1. 1: Se calcula el rayo original del método anterior, desde la cámara hasta el sensor.
2. 2: Se calcula la intersección de dicho rayo con el plano de enfoque, situado a la distancia `focusDistance`. La intersección se denomina `focusPoint`.
3. 3: En vez de emitir e 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.

```
#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
```

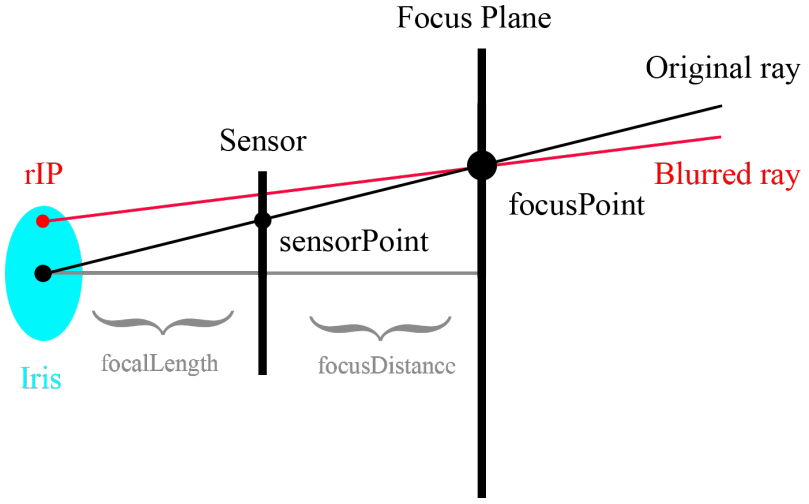


Figura 3.1

3.2. Iluminación

En la implementación básica solo se hacía uso de luz ambiental para iluminar la escena. Se puede hacer uso de distintos mecanismos de iluminación para dar mayor riqueza visual. A continuación se detallan los tipos de luces implementadas.

- Luces de punto:

  Las luces de punto son quizá el elemento más simple de iluminación. Consisten en un punto sin volumen el cual emite radiación lumínica de manera uniforme. Esta

radiación se desvanece de manera cuadrática. Debido a que son puntos infinitesimalmente pequeños, jamás serán alcanzados por los fotones emitidos desde la cámara, de manera que requieren un procesamiento especial explicado posteriormente en ??sec:mi]Muestreo por importancia.

Una luz de punto viene definida por tres atributos: Color, intensidad y posición. La energía lumínica viene dada por la ecuación:

- Materiales emisivos:

  Cuando un objeto es alcanzado por un rayo, lo más común es que se reste energía a dicho rayo debido a la absorción del material. Otra opción es sumarle energía. Si esto ocurre, se pasará a considerar a ese objeto como otra fuente de luz más.

  La energía sumada a dicho rayo será obtenida de una textura denominada emisión del material a través del mapeo correspondiente del punto de intersección, multiplicada así por un factor de intensidad.

- IBL (Image Based Lightning):

  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 y debido a que 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 (Imagen de Alto Rango Dinámico). 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. Esto es debido a que su uso no es meramente 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.

  La implementación de esta técnica en el motor de render viene dada por el uso de una textura en formato .hdr (imágenes en punto flotante), o un color plano. En caso de utilizar una textura, se considerará cada píxel como una pequeña fuente de luz direccional en el infinito, orientada hacia el centro de la escena.

  Los rayos que no interseccionan con nada se consideran que interseccionan en el infinito con el HDRI, por esta razón, al detectar que un rayo ha terminado de rebotar y ha terminado en el infinito, se obtendrá la dirección de este, y esta dirección se traducirá en las coordenadas polares del HDRI que posteriormente recuperarán el valor interpolado bilinearmente del pixel del HDRI correspondiente.

3.3. 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 cabezera. 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.

3.3.1. 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 comprendidas 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.

```
__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;
}
```

Debido a la naturaleza esférica de los mapas de entorno, resultará útil añadir dos funciones que transformen coordenadas esféricas en coordenadas `u,v`. Estas dos funciones son `sphericalMapping` y su inversa `reverseSphericalMapping`. 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`.

```
__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);
    }
```

```
    __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);
    }
```

3.3.2. Filtrado

Las texturas cuentan con valores discretos y resoluciones limitadas. Esto provoca que la imagen se pixelice cuando se muestra cercana a la cámara. Una solución adoptada de manera general en muchos ámbitos es la interpolación los píxeles vecinos. Actualmente muchos visualizadores de imágenes no muestran los píxeles naturales si no una versión modificada de estos. Esto se conoce como filtrado. En la implementación se ha usado un filtrado lineal conocido como interpolación bilinear. 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 pixel 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, además de introducir

3.3.3. Mapas de normales

Los mapas de normales son un recurso muy utilizado en programación gráfica, aún así no ha sido posible encontrar explicaciones detalladas para su implementación en bilbiografía orientada específicamente a Path Tracing. Por ejemplo el libro PBRT[5] no incluye mucha

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.

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 utilizará 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 pueden 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. Las normales suministradas por los mapas de normales son normales locales, en el espacio tangencial. El espacio tangencial es un espacio formado por la normal calculada a partir del triángulo, la tangente y la bitangente, tres vectores casi ortogonales (posteriormente se aclarará este çasi").

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: `Vector3 worldNormal = (localNormal.x * tangent - localNormal.y * bitangent + localNormal.z * normal).normalize`

3.4. Smooth Shading

Anteriormente en Intersección triángulo - rayo se explicaba como 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 sencillo arreglo 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 siguente:

```
#if SMOOTH_SHADING

    // https://gist.github.com/pixnblox/5e64b0724c186313bc7b6ce096b08820
```

```
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 a partir de las coordenadas baricéntricas del triángulo `u,v` las cuales indican la distancia a cada vértice.

Las coordenadas `u,v` son las coordenadas

3.5. Sombreado BRDF

El sombreado 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 $f_{\mathrm{r}}(\mathbf{x}, \omega_{\mathrm{i}}, \omega_{\mathrm{o}})$. Hasta ahora ha sido ignorado y simplificado como una superficie Lambertiana perfectamente difusa, esto quiere decir que hasta el momento 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]

El término "materialz "BRDF.están fuertemente ligados, ya que es el material el que define esta función a partir de sus parámetros.

Un elemento clave del renderizado fotorrealista es elegir una función de sombreado apropiada. En este motor se ha hecho uso del modelo de sombreado Disney Principled Shader. Este modelo fue desarrollado por Disney bajo el fin de simplificar los parámetros de las fórmulas matemáticas y que estos sean cómodos para los artistas. Esta decisión tiene más sentido si consideramos el contexto histórico, donde los modelos anteriores contaban con parámetros complejos.

Este modelo cuenta además con buen fotorrealismo, y por ello, el conocido software de edición 3d de código abierto Blender, hace uso de él como su modelo de sombreado primario.

A continuación se muestra una lista con los parámetros de los materiales descritos bajo este modelo:

roughness: metallic: clearcoatGloss: clearcoat: anisotropic: eta: specular: specularTint: sheenTint: subsurface: sheen:

Capítulo 4

Optimizaciones estructurales

Un algoritmo con buenos resultados visuales viene con 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 rayo cargue con información "más útil". Este procedimiento se denomina "Muestreo por importanciaz 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.

Así pues este término es bastante general. 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 técninas 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 muestrearse la mitad, se obtiene la mitad de radiación.

En la implementación, todo muestreo consiste en una función que genera rayos aleatorios a aquellos lugares donde interesa más muestrear y la función de probabilidad de distribución apropiada derivada de la función de muestreo, que ha de devolver la probabilidad con la que se ha elegido ese rayo. 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 entorno

Anteriormente se ha descrito como la iluminación HDRI permite resultados visualmente complejos de manera simple y sin necesidad de primitivas, simplemente una imagen con 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 tales como el sol el cual 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". 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.

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 (8K) 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 hasta el momento de cada pixel normalizada. 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 además 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).

Así pues, 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.

La función de distribución de probabilidad vendrá entonces dada por la iluminación para el pixel obtenido dividida por la iluminación total.

4.1.2. 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 el cual emite luz de manera uniforme hacia todos los lados. Sin la implementación de la estimación

de evento próximo, estas luces al ser infinitesimalmente pequeñas, jamás serían alcanzadas por los fotones y por lo tanto, nunca se recibiría información de ellas. 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.

4.1.3. Muestreo por importancia de BRDF

La función BRDF aporta distintas intensidades lumínicas dependiendo de la dirección entrante y la dirección saliente, así pues 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 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. Así pues, no tiene sentido emitir rayos aleatorios donde se conozca que el resultado del BRDF será 0.

Otro ejemplo es el muestreo por coseno. La ecuación de renderizado cuenta con un término que pondera radiación obtenida por el modelo BRDF a partir del coseno del ángulo incidente. Por esta razón, tiene sentido emitir de manera proporcional a este término más rayos allá donde el coseno es más grande.

Muestreo por importancia múltiple:

El muestreo por importancia es una herramienta que mejora considerablemente los tiempos de convergencia en determinados escenarios, aunque 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). Esta técnica evaluará todas las funciones de muestreo para cada intersección y dará mayor importancia con un escalar a aquellas funciones que proporcionen mayor información de la escena.

(AÑADIR TEORÏA AQUÍ)

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

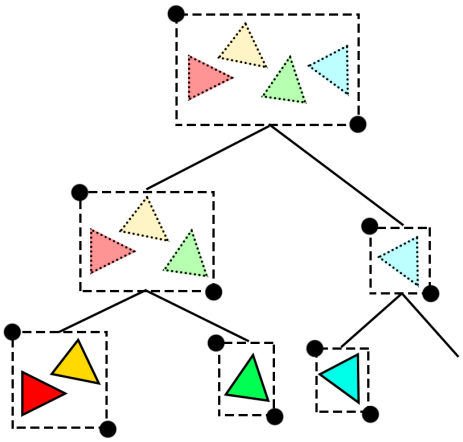


Figura 4.1: División BVH

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.

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

4.2.1. Operación de generación de BVH (CPU)

La operación de generación parte de la generación recursiva de un árbol binario, 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 que 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.

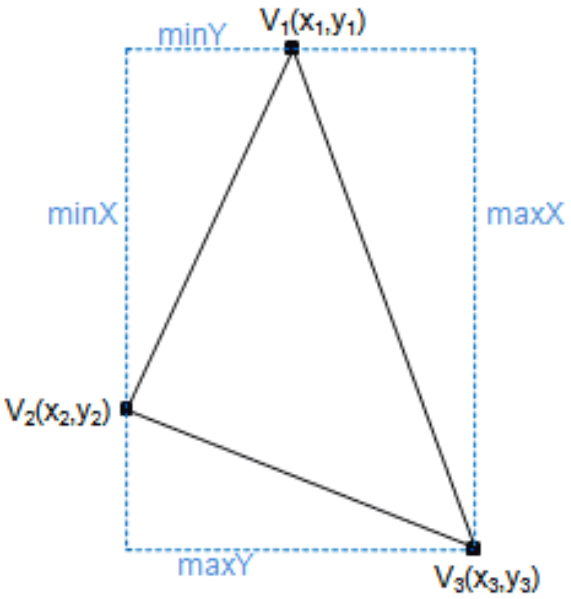


Figura 4.2: Delimitadores

```
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.

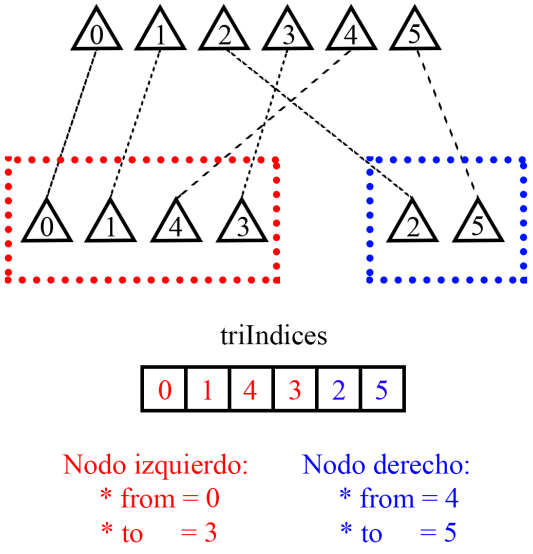


Figura 4.3: triIndices

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:

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)

La operación de recorrido implementada se resume en comprob

4.2.3. Evaluación:

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

Figura 4.4: Standford Bunny.

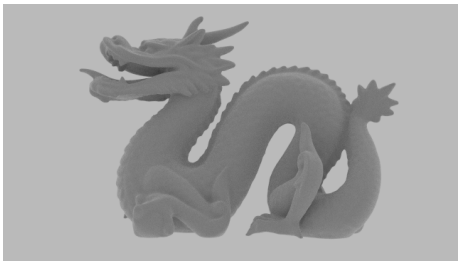
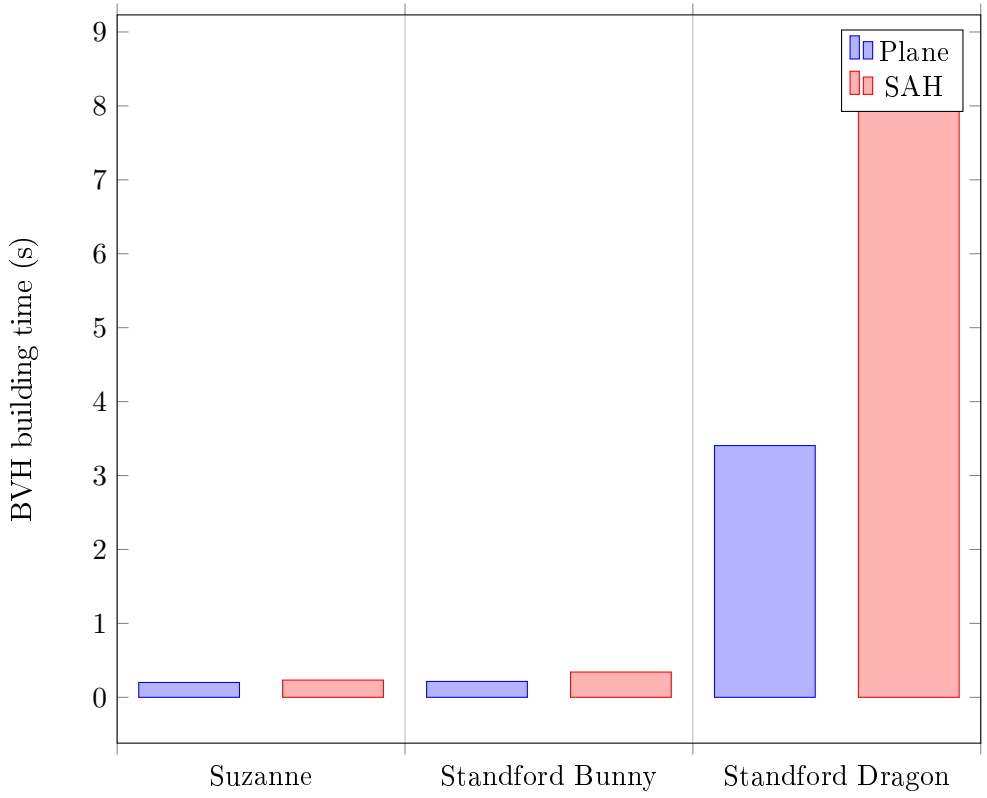


Figura 4.5: Standford Dragon.

Esta evaluación ha sido llevada acabo con los siguientes parámetros:
`BVH_DEPTH = 20`, `BVH_SAHBINS = 12`, `MAXBOUNCES = 5`, `SMOOTH_SHADING = true`,



En primer lugar se muestra

4.2.4. Selección de parámetros

Habiendo definido los métodos de construcción y recorrido, el siguiente paso es comprobar para qué parámetros de profundidad del árbol y de bining se obtienen los mejores resultados.

Capítulo 5

Port OneApi

Capítulo 6

Conclusiones

Capítulo 7

Anexo

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] James T Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.

[3] Jean-Henri Lambert. *JH Lambert,... Photometria, sive de Mensura et gradibus luminis, colorum et umbrae*. sumptibus viduae E. Klett, 1760.

[4] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of graphics tools*, 2(1):21–28, 1997.

[5] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.