UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIA DE LA COMPUTACIÓN

COMPARACIÓN DE MÉTODOS BASADOS EN BOUNDING VOLUMES HIERARCHIES PARA RAY TRACING EN ESCENAS DINÁMICAS

TESIS

Para optar el título profesional de Licenciado en Ciencia de la Computación

AUTOR:

Jonathan Andres Loza Mendoza

ASESORES

PhD(c) Luciano Arnaldo Romero Calla

MSc Rommel Quintanilla Cruz

Lima - Perú

2023

UTEC
UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

Dirección de Investigación

DECLARACIÓN JURADA

Yo, Mónica Cecilia Santa María Fuster identificada con DNI No 18226712 en mi condición de autoridad responsable de validar la autenticidad de los trabajos de investigación y tesis de la UNIVERSIDAD DE INGENIERIA Y TECNOLOGIA, DECLARO BAJO JURAMENTO:

Que la tesis denominada "COMPARACIÓN DE MÉTODOS BASADOS EN BOUNDING VOLUMES HIERARCHIES PARA RAY TRACING EN ESCENAS DINÁMICAS" ha sido elaborada por el señor Jonathan Andres Loza Mendoza, con la asesoría de Luciano Arnaldo Romero Calla, identificado con el DNI N°46725621 y de Rommel Anatoli Quintanilla Cruz identificado con el DNI N°45157275, y que se presenta para obtener el grado de Licenciado en Ciencia de la Computación, ha sido sometida a los mecanismos de control y sanciones anti plagio previstos en la normativa interna de la universidad, encontrándose un porcentaje de similitud de 0%.

En fe de lo cual firmo la presente.

Dra. Mónica Santa María Fuster
Directora de Investigación

En Barranco, el 27 de junio de 2023

Dedicatoria:

A mis padres, por su lucha y sacrificio. Su apoyo y confianza me ha acompañado desde que tengo uso de razón. Para ellos, porque me aguantan y me motivan

Agradecimientos:

Quiero agradecer a mis profesores y compañeros de mi época universitaria, por todas las experiencias y aprendizajes que han enriquecido mi vida. Han sido parte fundamental en mi desarrollo tanto profesional como personal

Índice general

Pág.

Índice de tablas

Índice de figuras

RESUMEN

Una de las técnicas de renderización más populares es el *Ray Tracing* el cual se basa en el funcionamiento de la luz. Este método evalúa qué objetos de la escena son intersectados por los haces de luz para determinar si son visibles. Para optimizar esta búsqueda se utilizan estructuras de datos, las cuales encapsulan los objetos de las escenas acelerando la búsqueda de intersecciones entre objetos y los rayos de luz.

Hacer uso de este método en escenas dinámicas, escenas en donde puede cambiar la posición o topología de los objetos, requiere que las estructuras tengan una construcción y un recorrido veloz. En consecuencia, nuevas propuestas basadas en las estructuras *Bounding Volume Hierarchies (BVH)* han sido presentadas, las cuales buscan disminuir el tiempo de construcción sin afectar el rendimiento en el recorrido. Sin embargo, existen pocos trabajos que comparen estas nuevas propuestas de *BVH* para evaluar su desempeño en escenas dinámicas.

El presente trabajo realiza la comparación de 3 estructuras de aceleración del estado del arte: *LBVH*, *KBVH* y *PLOC* utilizando escenas dinámicas. De esta comparación se calculó la cantidad de *frames* por segundo que son capaces de alcanzar cada estructura así como su calidad, utilizando la *Surface Area Heuristic*. Adicionalmente, las estructuras fueron comparadas en base a los tiempos de construcción y renderización en las diferentes escenas.

Palabras clave:
Ray tracing; Estructuras de datos; Escenas dinámicas

ABSTRACT

COMPARISON OF BOUNDING VOLUME HIERARCHIES-BASED METHODS FOR RAY TRACING IN DYNAMIC SCENES

Ray tracing is one of the most popular techniques used to render high-quality images. To optimize this method the 3D scenes are encapsulated inside a data structure that accelerates the search of ray-object intersections. In dynamic scenes is necessary that the structures have a fast build and traversal time. In response to these requirements, new proposals based on the Bounding Volume Hierarchies structure were presented, however few comparative papers have evaluated its performance in dynamic scenes.The present dissertation compares three different state-of-the-art acceleration structures: LBVH, KBVH and PLOC using a benchmark of dynamic scenes. For each structure, we evaluate the build time, render time, frames per second and measure the quality of the structure using the Surface Area Heuristic.

Keywords:

Ray tracing; Acceleration Data Structures; Dynamic scenes

Capítulo 1

INTRODUCCIÓN

En Computación Gráfica, la renderización es el proceso por el cual se generan imágenes a partir de escenas 3D. Parte fundamental de la renderización es poder determinar qué objetos son visibles desde la cámara, cómo les afecta la luz y qué efectos produce en la imagen final. Uno de los métodos más populares para el renderizado de imágenes es el *ray tracing* [1].

Esta técnica de renderización simula el funcionamiento de la luz para replicar sus efectos. Las diferentes fuentes de luz, como el sol, emiten fotones que al chocar con un objeto son redirigidos hacia nuestros ojos permitiéndonos ver su color. *Ray tracing* representa estos fotones como rayos los cuales son disparados hacia la escena 3D, buscando la intersección más cercana entre un rayo y un objeto. Gracias a esta técnica es posible obtener imágenes con altos niveles de realismo [2].

Sin embargo, basarse en este fenómeno físico tiene una desventaja ya que consume demasiado tiempo para poder obtener imágenes realistas de alta calidad [3]. Esto se debe principalmente a que debemos evaluar cada rayo con todos los objetos de la escena para verificar si este rayo lo intersecta o no, lo cual crece de forma exponencial según agreguemos más rayos y/o objetos. Conseguir un resultado plausible requiere lanzar una cantidad considerable de rayos en la escena; de lo contrario, se obtienen imágenes con una alta frecuencia de ruido [2]. Para poder mejorar esta técnica las primitivas, partes básicas de la construcción de imágenes, se organizan en estructuras de datos [1][2]. De esta manera, se acelera el tiempo de consulta y se disminuye la cantidad de intersecciones rayo-primitiva.

En el caso de aplicar *ray tracing* para escenas estáticas, la investigación en estructuras de aceleración se enfocaba solamente en optimizar el recorrido de la estructura, ya que la construcción es realizada de manera offline por lo que no era un factor de vital importancia. Sin embargo, en el caso de escenas dinámicas esto no aplica, ya que en cada *frame* las primitivas cambian su posición, invalidando la estructura, por lo que era necesario reconstruirla de forma rápida [1]. Una de las estructuras de aceleración más populares para aplicar en escenas dinámicas debido a su fácil construcción y actualización es el *Bounding Volume Hierarchies* (*BVH*) [2].

El presente trabajo busca realizar una comparación entre las implementaciones del estado del arte de *BVH*: *LBVH*, *KBVH* y *PLOC* en escenas dinámicas para evaluar su desempeño.

El resto del documento es organizado de la siguiente manera: el Capítulo 2 desarrolla el marco teórico donde se presentan los conceptos de *ray tracing* así como la descripción de las estructuras a comparar. El Capítulo 3 presenta los trabajos previos realizados en el área, tanto trabajos comparativos como las experimentaciones que se realizan entre estructuras. El Capítulo 4 describe la propuesta a seguir para llevar a cabo las comparaciones, tomando en cuenta las diferentes escenas, así como la cantidad de rayos a utilizar. El Capítulo 5 muestra la experimentación, resultados y discusión de los mismos para evaluar el desempeño de cada estructura. Finalmente, las conclusiones finales y trabajos futuros en el Capítulo 6.

1.1 Motivación y Contexto

Hoy en día, el estudio de estructuras de datos de aceleración es muy popular en el campo de *ray tracing* al generar una gran cantidad de nuevas propuestas [2]. El poder optimizar su uso para escenas dinámicas permitiría explotar aún más sus capacidades en ámbitos como los videojuegos o el modelamiento molecular.

Por otro lado, los últimos avances en tarjetas gráficas enfocan parte de su desarrollo en tratar de optimizar esta técnica [4] por lo que es un tema de mucho interés para las compañías más importantes de tecnología. Sin embargo, su uso actualmente es muy dependiente de las últimas generaciones de hardware avanzado, por ende, el más costoso.

1.2 Planteamiento del Problema

Las estructuras de datos de aceleración enfocadas en la optimización de la intersección rayo-primitiva quedan obsoletas al aplicarlas en escenas dinámicas, ya que en cada *frame* distinto los objetos en la escena se mueven. *Kd-tree* durante mucho tiempo ha sido considerada la mejor estructura para *ray tracing*, pero al agregar el contexto de escenas dinámicas, *BVH* ha obtenido toda la atención de los investigadores en la última década [2].

Hoy en día, los trabajos comparativos no han podido determinar qué estructura de aceleración de *BVH* tiene mejor desempeño para *ray tracing* en escenas dinámicas. Esto se debe principalmente a que los trabajos comparativos no han llegado a evaluar las estructuras en escenas dinámicas. Adicionalmente, se comparan algoritmos antiguos por lo que nuevas propuestas no han sido evaluadas [5][6] [7].

1.3 Justificación

Comparar los resultados de las propuesta de *BVH* de distintos trabajos es una tarea complicada ya que en cada experimentación se utilizan diferentes escenas y el hardware es distinto [1]. Por lo que realizar un análisis comparativo en un mismo ambiente permite tener resultados más precisos para contrastar las propuestas de *BVH* en escenas dinámicas.

Por otro lado, tener una comparación entre las distintas estructuras *BVH* permite saber en qué tipos de escenas se desempeña mejor cada una. Esto es de utilidad para

elegir qué estructura de aceleración usar a la hora de implementar un *raytracer* de escenas dinámicas.

Finalmente, sirve como punto de partida para que futuras propuestas de *BVH* tomen en consideración las escenas dinámicas en sus experimentos a la hora de comparar su propuesta con otras.

Tomando en cuenta lo antes mencionado, esta investigación plantea la comparación entre 3 propuestas de *BVH* en escenas dinámicas, usando una misma arquitectura y *hardware* para poder sacar conclusiones más acertadas sobre estas estructuras.

1.4 Objetivos

El objetivo principal de esta tesis es comparar las estructuras de aceleración de *BVH* para *ray tracing* en escenas dinámicas. En base a esto, podemos definir los siguiente objetivos específicos:

1. Comparar 3 estructuras de aceleración de *BVH* (*LBVH*, *KBVH*, *PLOC*) en la construcción y renderización de escenas dinámicas.
2. Determinar en qué tipos de escenas se desempeña mejor cada unas de las estructuras comparadas.

Capítulo 2

TRABAJOS RELACIONADOS

En este capítulo se presentarán los diversos trabajos relacionados sobre las estructuras de aceleración para *ray tracing*. En primer lugar se presentan los distintos trabajos que han realizado comparación entre estructuras de datos. Se especifica qué estructuras se comparan, con qué escenas y que es evaluado de cada comparación. Posterior a esto se presentan las diversas propuestas que han sido presentadas para *BVH* y se detalla como se ha realizado la experimentación en estos trabajos. Finalmente se presenta de qué manera es evaluada la calidad de las estructuras.

2.1 Trabajos Comparativos

Diferentes trabajos han comparado estructuras de aceleración para ray tracing con el fin de determinar la más óptima según el contexto de sus experimentos. El trabajo de Vinkler et al. [8] plantea una comparación entre BVH y kd-tree específicamente en arquitecturas de muchos núcleos, como las GPUs. Se compararon un BVH y kd-tree, ambos construidos usando el costo SAH. Sus experimentos sólo se evalúan utilizando escenas estáticas y el tiempo de renderización. De los resultados, kd-tree realizó menos evaluaciones para encontrar la intersección y funcionó mejor para escenas más grandes y complejas mientras que BVH tuvo mejor desempeño en escenas de mediano tamaño.

Otros trabajos que sí tomaron en cuenta el tiempo de construcción son los presentados por Thrane et al. [6] y Ge [7]. El primero evaluó el desempeño de los algoritmos en sistemas multi-core como las GPU para el tiempo de renderización y la construcción fue realizada en CPU. Las estructuras comparadas fueron 2 implementaciones de *BVH*

propuestos por Kay et al. [9] y Goldsmith et al. [10] respectivamente, 2 algoritmos de recorrido en *kd-tree* propuestos por Foley et al. [11] y un *grid* uniforme. Los autores muestran cómo el *BVH* tiene muy buen desempeño comparado con otras estructuras usando el GPU. Por otro lado, el trabajo presentado por Ge realiza su comparación utilizando la librería NIH de Nvidia la cual proporcionaba 2 estructuras: *kd-tree*, *LBVH*. El autor buscó centrarse netamente en la comparación de las estructuras por lo que buscó alguna fuente o repositorio confiable que les proporcione los códigos. Este trabajo se enfocó más en evaluar factores como el tiempo de construcción, utilización del ancho de banda, performance de la memoria caché y eficiencia de la ramificación. Sin embargo, para ambos trabajos solo se utilizaron escenas estáticas.

En el siguiente trabajo presentado por Ize [5], el autor plantea diferentes propuestas de estructuras de datos en escenas estáticas y dinámicas. Centrándose en el caso de escenas dinámicas, se presentan 2 propuestas. En primer lugar, el uso de un *macrocell grid* que se reconstruye en cada *frame* y el uso de 2 *BVH* en paralelo. El primer *BVH* es el que se encuentra ejecutándose en la escena y para manejar las actualizaciones solo se reajusta sus *Bounding Boxes* mientras que de forma asíncrona se construye un *BVH* que reemplazará al otro *BVH* que va perdiendo calidad con cada reajuste. La investigación está mucho más centrada en la explicación y presentación de resultados de sus métodos propuestos y no se comparan nuevas propuestas del estado del arte.

Finalmente, el artículo publicado por Macedo y Rodrigues [12] realiza una comparación entre diversas estructuras de aceleración en un ambiente 3D animado enfocándose en las reflexiones de los objetos. Los autores tratan de expandir un trabajo previo dedicado a simular reflexiones realistas utilizando *ray tracing*. Por lo tanto, deciden evaluar está propuesta en un ambiente animado comparando diferentes estructuras de aceleración. Los autores hacen uso de la librería *Nvidia OptiX* la cual les proporciona toda un API para renderizar con *ray tracing*, así como estructuras de aceleración. Se usaron 4 estructuras de aceleración: un *BVH estándar*, un *MedianBVH* capaz de crear estructuras de mediana

calidad pero de rápida construcción, el *LBVH* que genera estructuras de baja calidad pero con el tiempo de construcción más rápido y un *TriangleKdTree* capaz de generar estructuras de alta calidad. Si bien este trabajo realiza sus comparaciones en escenas dinámicas, está mucho más enfocado en evaluar su propuesta para tratar las reflexiones realistas en tiempo real.

2.2 Optimizaciones para las estructuras de datos

De acuerdo con Li et al. [13], para poder evaluar el comportamiento de las estructuras de aceleración en *ray tracing* es necesario obtener su tiempo de construcción y de recorrido de rayos. Para el caso de escenas estáticas, el tiempo de renderización es el aspecto primordial que se busca optimizar. Sin embargo, Wald et al. [1] menciona que para el caso de escenas dinámicas el tiempo de construcción ya no puede ser ignorado, lo que dio paso a nuevas propuestas de *BVH* que toman en consideración ambos factores a la hora de realizar sus experimentos.

Las experimentaciones en estructuras de aceleración tienden a usar escenas distintas para evaluar sus propuestas y compararlas con otras, no obstante, las evaluaciones tienden a ser las mismas.

La evaluación más común es medir el tiempo de construcción en milisegundos de las estructuras en distintas escenas [14][15][16][17][18][19][20][21] Esto se debe principalmente a que en escenas dinámicas al haber cambios en la geometría de los objetos será necesario reconstruir la estructura, en caso ésta no tenga un método de actualización que no degrade la calidad.

Otro factor relevante a evaluar es la calidad de las estructuras. Para esto se utilizan métricas que predicen el recorrido de las estructuras, las cuales serán detalladas en la Sección 2.3. Trabajos como [14] [15] [20] [21][22][23] hacen esta evaluación en sus experimentaciones para comparar sus propuestas en diversas escenas.

La última evaluación más recurrente que pone a prueba las estructuras para *ray tracing* es el recorrido de los *rays*. Artículos como [14][15][21] evalúan la velocidad de recorrido en milisegundos. Otra forma de evaluar el recorrido es compararlo con el recorrido de una estructura de alta calidad [16][17][20]. Esto representa el tiempo de renderizado que toman las estructuras para diferentes escenas. Sin embargo, las escenas usadas son estáticas.

En artículos como [24] [23], los autores han evaluado sus estructuras en escenas dinámicas, determinando la cantidad de *frames* que son capaces de renderizar por segundo. Sin embargo, sólo llegan a evaluar su propuesta, no es comparada con otras estructuras, lo cual sí es común en artículos que solo evalúan el tiempo de construcción y recorrido [14] [15] [16] [19] [22] [18].

Trabajos como el propuesto por Li et al. [13] realizaron la comparación de estructuras de aceleración al tomar resultados de otros trabajos y compararlos con los resultados de su algoritmo propuesto. Para esto realizaron una conversión de los resultados de las otras investigaciones para que se asemejen a las características de la máquina en donde ellos realizaron sus experimentos. Esto se debe a que muchos trabajos no tienen publicados sus códigos para poder replicar las estructuras. El escenario ideal sería que se puedan ejecutar todas las estructuras en un mismo ambiente para tener resultados más precisos para la comparación.

2.3 Calidad de las Estructuras de Aceleración

La calidad de las estructuras está relacionada con la velocidad de recorrido, es decir, que tan rápido puede un rayo recorrer la estructura hasta encontrar el primer objeto con el que se intersecta. Encontrar la estructura de aceleración más óptima para una escena está catalogado como un problema *NP-Hard* [25]. Por este motivo, se utilizan predictores para poder encontrar de forma aproximada la mejor estructura para una escena.

El predictor más común utilizado hoy en día es el *Surface Area Heuristic (SAH)*. Esta heurística calcula que tan costoso es recorrer un nodo en base al área de su superficie, su funcionamiento es explicado en la Sección 3.3.3. *SAH* es normalmente usado en los trabajos a la hora de construir las estructuras para poder determinar que particionamiento de los nodos es más beneficioso, así como para comparar la calidad de diferentes estructuras [14] [15] [20] [21][22][23].

A pesar de ser la heurística más utilizada está lejos de ser perfecta. Trabajos como [25] critican a *SAH* como la métrica ideal para determinar la calidad de una estructura y predecir la velocidad de recorrido, proponiendo 2 nuevas métricas para solventar esta situación. Sin embargo, *SAH* sigue siendo utilizado para comparar la calidad de las estructuras, ya que es la métrica menos costosa de calcular comparada con nuevas propuestas.

Capítulo 3

MARCO TEÓRICO

En esta sección se presentan los conceptos fundamentales sobre *ray tracing* y las estructuras de datos de aceleración.

3.1 *Ray Tracing*

Es definido como una técnica de renderización de imágenes enfocado en la iluminación, capaz de crear una imagen 2D a partir de una escena 3D. El concepto básico de esta técnica apareció por primera vez en el artículo publicado por Appel [3], el cual se enfoca en dibujar las sombras de los objetos haciendo evaluaciones de rayos punto a punto entre objetos y la luz. Posteriormente, Whitted [26] presentó el algoritmo que hoy se conoce como *ray tracing* clásico, tomando como base lo presentado por Appel y agregando los conceptos de rayos de reflección y refracción. Para poder entender cómo funciona esta técnica es importante definir primero algunos conceptos.

En primer lugar, un rayo se define como un intervalo en una línea, y esta puede ser representada mediante 2 componentes: un origen O y una dirección d [27]. Basado en esto, nosotros podemos definir una función $P(t)$ que recibe un número real t (distancia) y nos devuelve un rayo (Ver Ecuación 3.1):

$$P(t) = O + td \tag{3.1}$$

Utilizando este concepto, uno puede disparar rayos desde un punto inicial y buscar la intersección con el objeto más cercano, si es que existe. A esto se le conoce como *Ray*

Casting. Estos objetos con los que se busca una intersección están compuestos por primitivas, las cuales son elementos básicos como triángulos, puntos, líneas que son usados para crear imágenes más complejas. Por lo tanto, cuando se busca la intersección de un rayo con un objeto, en realidad se busca con que primitiva del objeto es que se intersecta el rayo.

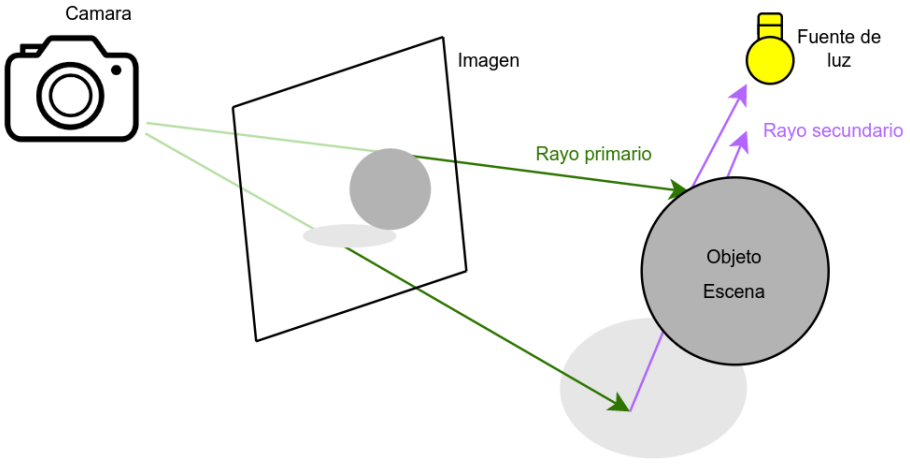

FIGURA 3.1: Funcionamiento de *Ray Tracing* haciendo uso de *Ray Casting*. Referencia de [27]

La Figura 3.1 representa el funcionamiento de *ray tracing*. Los rayos usados en esta técnica pueden ser divididos en 2 tipos: rayos principales, los cuales son los que se originan en la cámara; y los rayos secundarios los cuales son los rayos de sombra, de reflección y refracción, que se crean a partir de un rayo principal o secundario.

El procedimiento comienza disparando rayos principales desde la cámara hacia la escena. Estos se intersectan con las primitivas de la escena y a partir de ahí, disparan rayos en dirección a la luz, denominados rayos de sombra, para determinar si la zona de intersección rayo-primitiva es iluminada o no.

Además de los rayos de sombra también se pueden disparar rayos de reflección y refracción dependiendo del material del objeto. Este comportamiento se puede apreciar

en la Figura 3.2, donde a partir del rayo que llega a la esfera de cristal, se disparan otros 2, de reflección y refracción, los cuales pueden llegar a generar nuevos rayos y repetir el proceso.

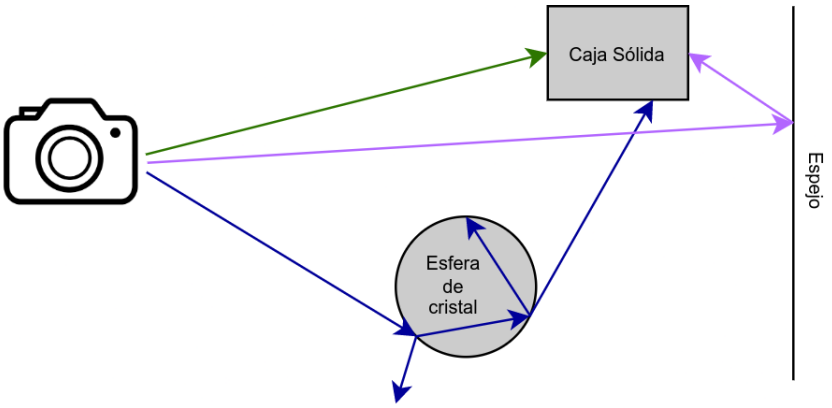

FIGURA 3.2: *Ray Tracing* usando rayos de reflección y refracción en la esfera de cristal y en el espejo. Referencia de [27]

Posteriormente, Cook [28] presentó el *ray tracing* estocástico, el cual consiste en disparar una mayor cantidad de rayos secundarios. El autor planteó esta propuesta ya que los efectos de sombras, reflección y refracción eran muy toscos, con su método se obtenian resultados mucho más suaves y realistas. Sin embargo, el uso de más rayos implicaba un mayor costo computacional.

3.2 Intersección de Rayos

La operación geométrica más básica de *ray tracing* consiste en encontrar la primitiva más cercana que se intersecta con un rayo. Determinar si existe o no una intersección se resume en una operación matemática basada en la geometría de la primitiva.

3.2.1 Rayo-Triángulo

Para poder determinar si un rayo intersecta a un triángulo, Moller et al [29] presentaron un algoritmo basado en las coordenadas baricéntricas, las cuales permiten representar un punto en un triángulo en base a 3 escalares w, u, v y sus 3 vértices V_0, V_1, V_2.

$$T(w, u, v) = wV_0 + uV_1 + vV_2 \tag{3.2}$$

La Ecuación 3.2 debe cumplir dos condiciones: $w, u, v >= 0$, $w + u + v = 1$. Esto nos permite definir w como $w = 1 - u - v$. Redefinimos nuestra función T en base solo a u y v:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3.3}$$

Con las condiciones de $u, v >= 0$ y $u + v <= 1$. Usando esta definición de punto en el triangulo nosotros podemos igualarla a nuestra función rayo $P(t)$, definida en la Ecuación 3.1, para determinar los valores de u y v.

$$P(t) = T(u, v)$$

$$O + td = (1 - u - v)V_0 + uV_1 + vV_2 \tag{3.4}$$

Reordenando los términos en formato de un sistema de ecuaciones obtenemos:

$$\begin{bmatrix} -d & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0 \tag{3.5}$$

Los autores resuelven el sistema usando la regla de Cramer, denotando $e_1 = V1 - V0, e_2 = V2 - V0$ y $T = O - V0$:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-d, e1, e2|} \begin{bmatrix} |T & e_1 & e_2| \\ |-d & T & e_2| \\ |-d & e_1 & T| \end{bmatrix} \tag{3.6}$$

Finalmente si los valores de $u, v >= 0.0$ y $v + u <= 1$ entonces podemos concluir que el rayo intersecta el triángulo (ver Ecuación 3.6).

3.2.2 Rayo-BV

Para escenas más complejas, se pueden diseñar objetos utilizando mallas triangulares. Sin embargo, determinar qué espacio ocupan estos objetos en la escena puede llegar a ser complicado dependiendo de qué tan complejos sean. Por esta razón, determinar si un rayo llega a intersectarlo se vuelve una tarea costosa ya que habría que evaluar cada una de sus primitivas.

Kajiya et al [9] propusieron encapsular los objetos en *bounding volumes (BV)* para solucionar este problema. Un *bounding volume*, también referido como *bounding box*, puede ser definido como una forma geométrica simple que contiene un objeto con una forma geométrica mucho más compleja. Tomando esto como base, el determinar si un rayo va a intersectar un objeto o no se reduce a evaluar si el rayo intersecta con el *BV*. Si

no lo intersecta, entonces no intersecta al objeto, por lo tanto, no será necesario evaluar sus primitivas y buscar posibles intersecciones.

Existen diversos tipos de *bounding volumes*, tales como:

- Esferas: objetos se encuentran contenidos en esferas.
- *Axis-aligned bounding box (AABB)*: objetos contenidos en una caja (box) con su lados alineados a los ejes de la escena.
- *Oriented bounding box (OBB)*: similares a los *AABB* pero sus lados no se encuentran alineados a los ejes de la escena.

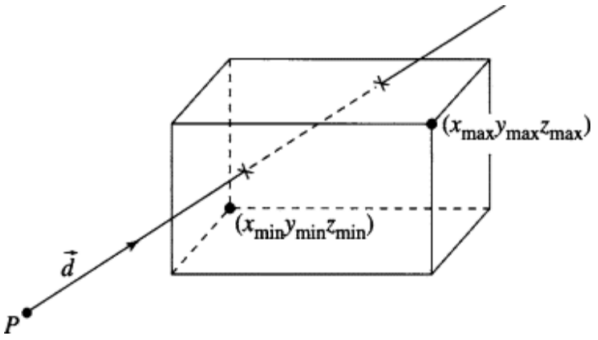

FIGURA 3.3: Axis-aligned Bounding Box intersectado por un rayo [30]

Los *BV* más usados en el caso de *ray tracing* son precisamente los *AABB*. Este *BV* puede ser definido usando 2 puntos que representan las coordenadas mínimas y máximas de cada eje, como se puede ver en la Figura 3.3. Para poder determinar si un rayo intersecta un *AABB* Smits [31] presentó el Algoritmo 1. Este algoritmo define unos intervalos mínimos y máximos en los que se pueden mover un rayo para luego usando la función del rayo poder determinar si este se encuentra dentro de los márgenes del *AABB*.

El Algoritmo 1 debe ser replicado para cada eje. Posteriormente, Williams et al [32] presentaron un artículo con una optimización a nivel de código y la implementación del algoritmo de Smits.

Algoritmo 1: Algoritmo para determinar si un rayo intersecta a un *AABB*. Los valores $minX$ y $maxX$ hacen referencia a los límites de un *AABB*. La condición de $invDir > 0$ sirve para determinar la orientación del rayo. La intersección existirá mientras no se rompa la condición de $intervaloMin > intervaloMax$. Se debe ejecutar para los 3 ejes [31]

```
intervaloMin ← ray.T.min;
intervaloMax ← ray.T.max;
t1 ← (minX − origenX) ∗ invDirX;
t2 ← (maxX − origenX) ∗ invDirX;
si invDirX > 0 entonces
    si t1 > intervaloMin entonces
        intervaloMin = t1;
    si t2 < intervaloMax entonces
        intervaloMax = t2;
    si intervaloMin > intervaloMax entonces
        devolver falso;
en otro caso
    si t2 > intervaloMin entonces
        intervaloMin = t2;
    si t1 < intervaloMax entonces
        intervaloMax = t1;
    si intervaloMin > intervaloMax entonces
        devolver falso;
devolver verdadero;
```

3.2.3 Problema de las intersecciones

El uso de los *bounding volumes* dentro de la escena reduce el costo computacional de evaluar un *ray* para cada primitiva de la escena. Un pseudocódigo se encuentra en el Algoritmo 2.

Como podemos observar, es un algoritmo sencillo, sin embargo para poder obtener un buen resultado del algoritmo es necesario lanzar muchos rayos tanto primarios como secundarios [2]. Además, solo usar *bounding volumes* para cada objeto de la escena puede no escalar de manera eficiente, ya que las escenas pueden contener objetos muy grandes y

Algoritmo 2: Algoritmos que evalúa todos los rayos de la escena para encontrar su intersección más cercana con una primitiva. Es costoso porque evalúa todos los rayos con todos los BV

```
para rayo en la escena hacer
    para BV en la escena hacer
        si rayo intersecta BV entonces
            para primitivas en el BV hacer
                si rayo intersecta a primitiva entonces
                    guardar la intersección mas cercana;
```

complejos. Por lo tanto, un *BV* tendría un objeto con muchas primitivas y el rayo tendría que evaluar cada una.

Este problema presentó la necesidad de poder almacenar las escenas en estructuras que optimicen esta intersección rayo-primitiva.

3.3 Estructuras de datos de aceleración

Las estructuras de datos de aceleración surgen con el propósito de reducir la cantidad de primitivas que un rayo debe evaluar en la escena. Estas estructuras transforman los datos de la escena a un formato que busca minimizar la cantidad de intersecciones que se deben evaluar durante la ejecución del algoritmo [33].

Las diferentes estructuras de aceleración usan dos enfoques para dividir la escena: subdivisión espacial y jerarquías de objetos. Las subdivisiones espaciales, como los *kd-trees* o *grids*, dividen la escena en los diferentes ejes y sus primitivas pueden ser referenciadas varias veces. Por otro lado, las jerarquías de objetos, como *Bounding Volume Hierarchies*, dividen la escena en base a las primitivas, por lo que solo tienen una referencia a cada primitiva, pero algunas de ellas pueden superponerse [1].

Teniendo en cuenta el algoritmo de recorrido, en la subdivisión espacial es más fácil encontrar la intersección más cercana ya que cada espacio de la escena solo es representado una sola vez. Sin embargo, como pueden existir diversas referencias a una misma primitiva esta se podría visitar más de una vez, cosa que no sucede en estructuras basadas en jerarquías de objetos.

Las subdivisiones espaciales pueden evaluar menos intersecciones con primitivas, pero el costo de construcción es mayor que las jerarquías de objetos. Además, es más difícil actualizar las implementaciones de subdivisiones espaciales cuando se utilizan en escenas animadas. Las jerarquías de objetos tienen ventaja a la hora de actualizarse ya que solo deben modificar el *Bounding Box* de las primitivas. Por estos motivos, para escenas dinámicas nos enfocaremos en los *Bounding Volume Hierarchies*.

3.3.1 *Bounding Volume Hierarchies (BVH)*

Son una estructura en forma de árbol con un factor de ramificación arbitrario, punteros a nodos interiores y referencias a las primitivas de la escena en las hojas del árbol. Cada uno de estos nodos representa un *BV* que contiene otros *BV*. Una característica que diferencia a los *BVH* de otras estructuras es que los *BV* usados en *BVH* pueden sobreponerse, evitando tener referencias múltiples a primitivas en diferentes nodos [2].

3.3.1.1. Construcción

Se pueden construir con un stack usando un método *top-down*. Se comienza ingresando el *root*, que contiene todas las primitivas de la escena, al stack. El algoritmo se ejecuta mientras el stack no esté vacío, haciendo un *pop* de un nodo del stack en cada iteración. Este nodo es evaluado en base a un criterio de terminación, que puede ser un

número máximo de primitivas, para decidir si se transforma a nodo hoja. De no cumplir con el criterio procede a dividirse en k nodos y son agregados al stack y se repite el proceso[2]. Un pseudocódigo se muestra en el algoritmo 3

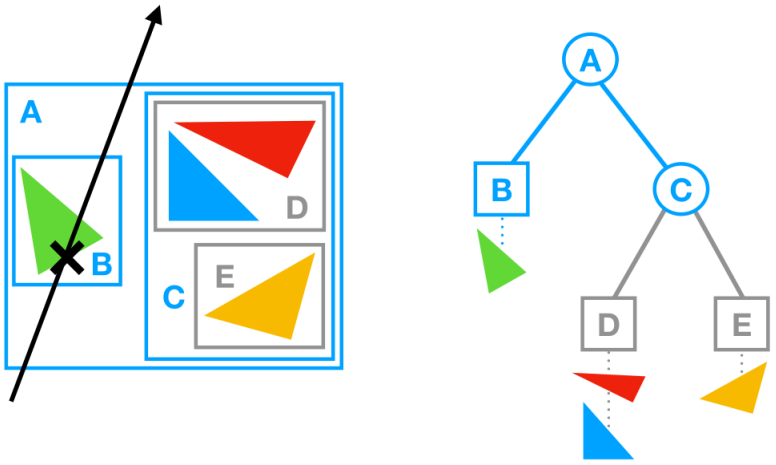

FIGURA 3.4: Representación de un *BVH* intersectado por un rayo. El rayo primero intersecta al *BV* A que contiene toda la escena. Luego evalúa a sus nodos interiores, que serían el *BV* B y C. Como el rayo no intersecta al *BV* C, entonces todo lo contenido en C no es evaluado. Por otro lado, en el *BV* B al ser un nodo hoja se evalúan todas sus primitivas con el rayo [2]

El método usado para elegir de qué forma dividir los nodos influye directamente en la calidad del árbol y en el tiempo requerido para recorrerlo [1].

3.3.1.2. Recorrido

El algoritmo básico para recorrer un *BVH* toma como base el mismo criterio usado por los *BV*. Si un *ray* intersecta un *BV*, entonces procederá a evaluar los *BV* que contiene en su interior, como se puede ver en la Figura 3.4. Si un *BV* no es intersectado por el rayo, entonces sus nodos interiores no tendrán que ser evaluados, como se puede ver en la Figura 3.4. El Algoritmo 4 representa este funcionamiento [2].

Algoritmo 3: Algoritmo convencional para construir un *BVH*. La función *particionamientoFinal* sirve para evaluar si un $nodo$ cumple con las condiciones para ser una hoja y ya no se debe particionar. De no cumplir, se particiona en 2 hijos y estos se agregan al stack para que sean evaluados [2]

```
Función construirBVH(raiz R):
    raíz contiene todas las primitivas de la escena;
    S ← Stack;
    S.push(R);
    mientras S no esta vacio hacer
        nodo ← S.pop;
        si particionamientoFinal(nodo) entonces
            nodo es setado como hoja;
        en otro caso
            dividir las primitivas del nodo en hijos;
            asignar hijos a nodo como su nodos interiores;
            para hijo en nodo hacer
                S.push(hijo);
```

3.3.2 Variaciones de *BVH*

La razón principal para utilizar estructuras de datos de aceleración es reducir el número de intersecciones rayo-primitiva. Sin embargo, el rendimiento del recorrido de rayos está muy relacionado con la calidad de la estructura de datos. La compensación entre estos dos procesos es invertir más tiempo en la construcción de las estructuras queda como resultado una estructura de alta calidad que reduciría el tiempo necesario para ejecutar el recorrido del rayo. Por otro lado, invertir menos tiempo en la construcción conduce a estructuras de baja calidad que lleva más tiempo recorrer [2][1]. Tomando en consideración esto, se han propuesto diferentes variaciones de *BVH* para optimizar la construcción sin perjudicar el recorrido, con el fin de utilizarlos en escenas dinámicas.

Algoritmo 4: Algoritmo para recorrer un *BVH*. Usa un stack para almacenar los nodos que debe evaluar. Si el rayo intersecta al nodo entonces se evalúa, sino se descarta. Solo se evalúan las primitivas cuando se llega a un nodo hoja [2]

```
Función recorrerBVH(raíz N, rayo R):
    S ← Stack;
    S.push(N);
    mientras S no esta vacio hacer
        nodo ← S.pop;
        si R intersecta a nodo entonces
            si nodo no es una hoja entonces
                para hijo en nodo hacer
                    S.push(hijo);
            en otro caso
                para primitiva en nodo hacer
                    evaluar si R intersecta a primitiva;
```

3.3.2.1. *LBVH*

El método *Linear Bounding Volume Hierarchy* (*LBVH*) fue propuesto como un algoritmo que reducía la complejidad de la construcción de un *BVH* a un problema de ordenamiento [16]. Para esto hacía uso de los *Morton Codes* para ordenar las primitivas de la escena y usar esto como base de la construcción de la estructura.

El primer paso de este algoritmo es realizar el ordenamiento de las primitivas de la escena. Para esto se encapsulan cada una de las primitivas en un *AABB* y se les calcula el baricentro. Este valor será el punto representativo de cada primitiva. Posterior a esto, se construye una rejilla tridimensional de tamaño $2^k \times 2^k \times 2^k$ que engloba toda la escena. Utilizando el punto representativo podemos determinar a qué rejilla pertenece cada primitiva. Con esto podemos representar cada una de las primitivas con un *Morton Code* de tamaño *3k-bits*. Este código se calcula al intercalar cada uno de los bits de las 3 coordenadas del punto representativo. Los autores hacen uso del algoritmo *radix sort*

paralelo para ordenar las primitivas, lo que genera una lista ordenada de primitivas en base a la curva de *Morton*.

Para construir la estructura a partir de la lista se deben crear los intervalos para cada nodo. Por ejemplo, asumiendo que el número de primitivas es n, entonces la raíz de la estructura tendría el intervalo de $[0, n)$ y sus hijos tendrían el intervalo $[0, m)$ y $[m, n)$ y así para cada nivel. Por lo que el problema se resume a encontrar estos puntos m donde se particionan los nodos. El algoritmo crea una lista de particionamiento en donde cada primitiva i calcula de manera paralela con su adyacente $i+1$ cuál es el bit más significativo en el que sus 2 *Morton Code* difieren. Asumiendo que la posición donde los bit difieren es en h, entonces se crea el par (i, h) el cual es agregado a la lista de particionamiento. El valor i se refiere al índice de la primitiva en la lista y h el nivel del árbol en donde ocurre el particionamiento. Una vez se tiene la lista esta se ordena en base al 2do valor de los pares. Con esto, tenemos la lista de particionamiento ordenada de tal forma que se realice el particionamiento comenzando desde el nivel 0, es decir desde la raíz, y así hasta llegar a las hojas donde se encontrarán las primitivas.

Finalmente, usando la lista de particionamiento ordenada, se procede a construir la estructura *BVH* en base a los pares (i, h). La posición i determinará en qué posición de la lista de primitivas se realizará el particionamiento, creando 2 nodos. En este proceso se construye el *BVH*, agregando los punteros a los hijos y dándole la estructura normal de estructura de datos para que luego pueda ser recorrida por un rayo.

3.3.2.2. *KBVH*

Karras et al. [21] propusieron un método para optimizar la calidad de los *BVH*. Ellos detectaron que los algoritmos utilizados para construir de manera rápida los *BVH* generaban árboles de muy mala calidad. Para esto, presentaron su técnica de particionamiento de triángulos la cual busca disminuir el tamaño de los triángulos para obtener

una distribución más uniforme y optimizar desempeño de *ray tracing*. El resultado del particionamiento de triángulos es después usado como el *input* para la construcción del *BVH*.

Lo primero es definir un parámetro β (porcentaje de particionamiento) con el cuál se determinará el número máximo de particiones que se realizará en una escena. Esto se calcula con la fórmula $s_{max} = \lfloor \beta.m \rfloor$, siendo m el número de triángulos en la escena. De esta forma, el algoritmo puede configurar una mayor cantidad de particiones a costo de un mayor tiempo de construcción.

Para cada uno de los triángulos en la escena, se calcula de forma paralela una heurística de prioridad p_t (t como índice de cada triángulo). Esta heurística sirve para determinar qué tan relevante es particionar un triángulo o no. Con esto los autores pueden determinar cómo distribuir la cantidad de particiones que deben hacer a lo largo de toda la lista de triángulos. Para esto, se define el número de particiones por triángulo como s_t el cual se calcula como $s_t = \lfloor D.p_t \rfloor$. En este caso D es un factor de escala que se busca maximizar mientras se siga cumpliendo la siguiente condición: $\sum s_t \leq s_{max}$.

Al finalizar este cálculo, cada triangulo tendrá un número de particionamiento independiente s que se tendrá que ejecutar. Para esto, se define un *stack* de "tareas de particionamiento". En el primer paso, el triángulo original entra al stack con su contador de particiones s. Se realiza el particionamiento y de este salen ahora 2 nuevos triángulos, por lo que habrá que calcular el contador de s_t para cada uno de las 2 particiones. Se denominan w_a y w_b como el valor del eje más grande para la partición A y la partición B, respectivamente. Con estos valores se definen los contadores s_a y s_b de la siguiente manera: $s_a = \lfloor (s-1)w_a/(w_a + w_b) + \frac{1}{2} \rfloor$ y $s_b = s - 1 - s_a$. Si el nuevo contador s_a es diferente de 0, entonces se agrega el *AABB* de la particion A junto a su contador s_a al *stack*. Lo mismo sucede para la partición B y su contador s_b si es que es diferente de 0. El proceso se repite hasta que el *stack* esté vacío. Con esto se realizan las particiones de

los triángulos y este vendría a ser el nuevo input de primitivas para un constructor *BVH*, como *LBVH*.

3.3.2.3. *PLOC*

Meister et al. [14] propusieron el método *Parallel Locally Ordered Clustering* (*PLOC*) para la construcción de *BVH* en arquitecturas de múltiples núcleos. Este algoritmo realiza un agrupamiento aglomerativo utilizando un ordenamiento basado en los *Morton Codes* para encontrar los vecinos más cercanos. Este se compone de 3 fases: búsqueda del vecino más cercano, fusión de grupos y una compactación.

Antes de empezar con la ejecución de las 3 fases. El algoritmo empieza por preparar el *input* inicial. En este se calculan los *BV* y el *Morton Code* de cada una de las primitivas de la escena. Usando los *Morton Codes* se ordenarán las primitivas y se considerará cada una de ellas como un grupo independiente. Este array ordenado de primitivas vendría a ser el *input buffer* inicial del algoritmo. Con esto se entrará al bucle principal, en el cual se encuentran las 3 fases, que se ejecutará hasta que solo quede un grupo en el *input buffer*.

Para la fase de búsqueda, los autores definen un parámetro r, el cual determina el rango de búsqueda para cada grupo. Si tenemos el grupo en la posición i del *input buffer*, este buscará a sus vecinos más cercanos en un rango de $i - r, i + r$. Para cada uno de los posibles candidatos se evaluará que tan cerca están en base a una función de distancia. Esta función recibe 2 grupos y calcula el área de la superficie del *BV* que encapsula a ambos grupos. Cada uno de los grupos guardará el candidato con el que tenga la menor distancia como su vecino más cercano. Como podemos observar, cada uno de los grupos puede calcular su vecino más cercano sin necesidad de los otros grupos, por lo que esta fase es altamente paralelizable.

En la siguiente fase, se decidirá qué grupos serán agrupados para continuar con el agrupamiento aglomerativo. La condición para que 2 grupos se junten es que ambos sean el vecino más cercano del otro. Es decir, para que el grupo G_1 se junte con el grupo G_2, el vecino más cercano de G_1 debe ser G_2 y el vecino más cercano de G_2 debe ser G_1. Al cumplirse esta condición los grupos se juntarán en la posición del primer grupo (el que tenga el índice i mas pequeño) y el otro grupo pasará a ser un grupo invalido.

Finalmente se realiza el proceso de compactación, en donde se busca eliminar los grupos inválidos y generar un *output buffer* que pasará a ser el nuevo *input buffer* en la siguiente iteración del bucle. Para esto se realiza una suma de prefijos paralela y exclusiva a los grupos, validando si son grupos activos o inactivos. De esta manera obtenemos las nuevas posiciones de los grupos en el *output buffer* y descartamos los grupos inactivos. Al terminar esto, se realiza un intercambio entre los valores del *output buffer* y el *input buffer* y se sigue con el bucle hasta que solo quede un grupo.

3.3.3 Heurística de Calidad

Como hemos visto, la parte fundamental de poder construir estructuras de buena calidad depende mucho de la heurística que tomemos a la hora de particionar nuestros *BV*. Los métodos más comunes han sido siempre utilizar media espacial o una media de objetos [1]. Sin embargo, estos dos métodos no toman en cuenta el costo que puede llegar a generar un particionamiento a la hora de recorrer el árbol.

Surface Area Heuristic (SAH) busca estimar el costo de recorrer un *BV* después de particionarlo. Para realizar esto hace uso del área de la superficie de un *BV* para determinar qué tan probable es que un *ray* intersecta el nuevo *BV* particionado [34].

En primer lugar, el *SAH* realiza algunas suposiciones: los *rays* son distribuidos uniformemente a lo largo de la escena; el costo de realizar un *traversal step* y una intersección

de un triángulo son valores conocidos determinados como C_t y C_i , respectivamente; el costo de intersectar N triángulos es de NC_i.

Usando estas suposiciones plantea la siguiente fórmula que calcula la probabilidad de que un *ray* que ha intersectado al nodo V_p tambien intersecte a V_c que es su nodo hijo:

$$P(V_c|V_p) = \frac{SA(V_c)}{SA(V_p)} \tag{3.7}$$

La Ecuación 3.7 calcula el área de la superficie (*SA*) de cada *bounding volume*. Como se puede apreciar en la Figura 3.5, mientras más grande sea el área de la superficie del nodo hijo, mayor será la probabilidad de que el *ray* que ha intersectado a su nodo padre también lo intersecte.

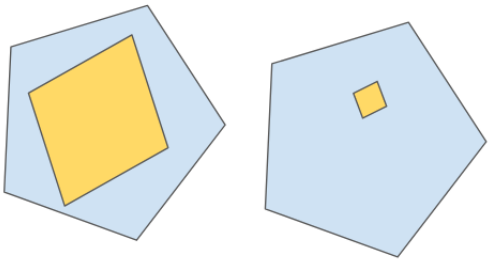

FIGURA 3.5: Mientras más grande sea el área del nodo interior, mayor será la probabilidad de que un rayo lo intersecte

Finalmente, la Ecuación 3.8 representa el costo de particionar el *BV* en el hiperplano p.

$$C(p) = C_t + P(L|V)N_L C_i + P(R|V)N_R C_i \tag{3.8}$$

Para calcular el costo de particionamiento se suma el costo de recorrido con la probabilidad P de ambos nodos hijos L y R, multiplicado por la cantidad de primitivas en cada nodo y el costo de intersección.

3.4 Escenas Dinámicas

De acuerdo con [1], hay muchas consideraciones que todo investigador debe evaluar con respecto al uso de estructuras de datos de aceleración para escenas dinámicas.

En primer lugar, no podemos encapsular todo el concepto de escenas dinámicas en solo una variación de la geometría de una primitiva. Las escenas dinámicas se pueden dividir en:

- Escena estática: la geometría de la escena no cambia entre *frames*.
- Escena parcialmente estática: algunas partes de la escena no cambian mientras que otras primitivas cambian su geometría.

Otro concepto para diferenciar las escenas dinámicas es en base al movimiento de las primitivas:

- Movimiento Jerárquico: donde las primitivas del mismo objeto tienen la misma transformación lineal.
- Movimiento incoherente: cada primitiva tiene su propia transformación.
- Movimiento semi-jerárquico: una escena híbrida que contiene objetos con movimiento jerárquico o incoherente.

La topología de la escena también puede verse afectada por estas variaciones en la escena animada. Una transformación regular que mantiene la topología de la escena solo mueve los vértices de los triángulos en una malla triangular pero mantiene la conectividad de estas primitivas. Por otro lado, una escena deformable cambia la topología de la escena variando la conectividad de la malla triangular.

3.4.1 Almacenamiento de escenas dinámicas

Una forma de almacenar las escenas dinámicas para luego ser renderizadas es guardar cada uno de sus frames en archivos separados que contengan la estructura de los objetos de la escena.

3.4.1.1. Polygon File Format (PLY)

Es un formato de archivo 3D que permite almacenar mallas tridimensionales al guardar la información de la posición de cada uno de sus polígonos. Este tipo de archivo guarda todos los vértices que se encuentran en la escena según su posición x,y,z y luego construye los polígonos al especificar la construcción de caras utilizando los vértices previamente definidos. Adicionalmente, se pueden almacenar información como las normales de la superficie, coordenadas de texturas y transparencia. El archivo se divide en 2 partes principales, la cabecera donde se encuentra la información de los datos que se van a otorgar y el cuerpo donde ya están los datos como los vértices y las caras. Un ejemplo de archivo PLY es el siguiente:

```
ply
format ascii 1.0    (formato del archivo: ascii o binario)
element vertex 8    (indica que se proporcionarán 8 vértices)
property float x    (la coordenada x es de tipo flotante)
property float y    (la coordenada y es de tipo flotante)
property float z    (la coordenada z es de tipo flotante)
element face 6      (indica que se definirán 6 caras con los vértices)
property list uchar int vertex_indices (vertex_indices tipo INT)
end_header          (final de la cabecera)
0.0 0.0 0.0         (información del primer vértice en orden x y z)
0.0 0.0 1.0
0.0 1.0 0.0
0.0 1.0 1.0
```

```
1.0 0.0 0.0
1.0 0.0 1.0
1.0 1.0 0.0
1.0 1.0 1.0
4 0 1 3 2           (información de la primera cara con 4 vértices)
4 6 7 1 0
4 1 7 5 3
4 6 0 2 4
4 2 3 5 4
4 4 5 7 6
```

3.4.1.2. Wavefront OBJ

De manera similar, los archivos Wavefront OBJ almacenan información de objetos tridimensionales. Permiten representar la información de la posición de cada vértice, sus normales, vértices de texturas y caras poligonales. En cada línea se especifica primero que tipo de dato se va a representar. Los vértices empiezan con la letra "v" y luego sus coordenadas x, y, z en números flotantes. De manera similar se pueden representar la información de las normales y los vértices de textura, solo que en este caso la letra inicial cambia, siendo "vn" para las normales y "vt" para las texturas. Finalmente, se presentan las caras con la letra "f" con los índices de cada vértice separado por un espacio. Adicionalmente se pueden relacionar cada vértices con un índice de coordenada de textura y normal separándolos por un "/". Un ejemplo de archivo OBJ es el siguiente:

```
v 0.0 0.0 0.0       (vértice con las coordenadas x,y,z)
v 1.0 0.0 0.0
v 0.0 1.0 0.0
v 0.0 0.0 1.0

vt 0.0 0.0          (coordenada de textura u,v )
vt 1.0 0.0
```

```
vt 1.0 1.0
vt 0.0 1.0

vn 0.0 0.0 -1.0     (normal de vértice con las coordenadas x,y,z)
vn 0.0 -1.0 0.0
vn -1.0 0.0 0.0

f 1/1/1 2/2/1 3/3/1 (cara con formato v/vt/vn)
f 1/1/2 3/3/2 4/4/2
f 1/1/3 4/4/3 2/2/3
f 2/2/1 4/4/1 3/3/1
```

Capítulo 4

PROPUESTA

En este capítulo se describirá la metodología propuesta para realizar una experimentación que nos permita comparar las diferentes estructuras de aceleración para escenas dinámicas.

En este trabajo se plantea comparar 3 estructuras del estado del arte de *BVH* para *ray tracing* en escenas dinámicas. Como se mencionó anteriormente, *BVH* ha ganado mucha fuerza en los últimos años generando mucha investigación como estructura de aceleración, debido principalmente a su fácil construcción y adaptación. Por este motivo, existen muchas propuestas interesantes para poder evaluar y sacar nuevas conclusiones.

Las estructuras seleccionadas son: *PLOC* [14] , *LBVH* [16] y *KBVH* [21]. Al buscar estructuras idóneas para escenas dinámicas es necesario que estas sean capaces de reconstruirse de manera muy veloz.

A día de hoy, *LBVH* podría ser considerado uno de los algoritmos más veloces de *BVH*, sin embargo genera árboles de baja calidad, por lo que sería interesante observar cómo se comporta la estructura al construir a mucha velocidad pero teniendo una recorrido ineficiente [2]. Por otro lado, *KBVH* es un método de optimización que particiona las primitivas de la estructura para mejorar su calidad y mejorar el recorrido del árbol. Finalmente, *PLOC* se enfoca en paralelizar altamente su construcción, por lo que es un algoritmo bastante rápido pero generando estructuras de mediana-alta calidad. Estas estructuras basan su construcción en métodos *bottom-up* lo cual los hace muy eficientes según crezca la cantidad de threads que se puedan usar. Finalmente, todas estas estructuras tienen sus códigos publicados lo cual asegura la calidad y el funcionamiento correcto de los algoritmos.

4.1 Metodología

Como mencionamos en 3.3.2, las diferentes estructuras de aceleración están constantemente tratando de aumentar su velocidad de construcción sin perder la calidad de su estructura, lo cual perjudica el recorrido. Por esta razón, tendremos en consideración estos dos aspectos a la hora de realizar nuestros experimentos.

La primera métrica a evaluar es la velocidad de construcción de las estructuras en las diferentes escenas. Posteriormente, se requiere evaluar la cantidad de *frames* que son capaces de renderizar en 1 segundo (FPS). Esta evaluación tiene que ser realizada en diferentes resoluciones para observar cómo se comportan con un número mayor o menor de píxeles, que significa una mayor cantidad de rayos en la escena. Se evaluará en 3 resoluciones, 854x480, 1920x1080 y 3840x2160. Finalmente, calcularemos el costo *SAH* de cada estructura en las diferentes escenas.

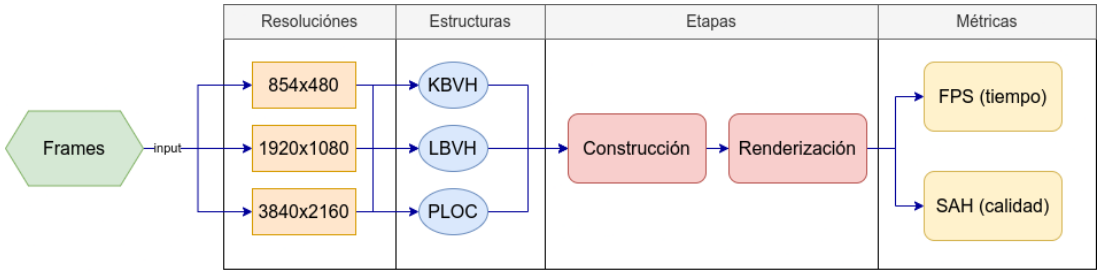

FIGURA 4.1: Metodología para evaluar el desempeño de las estructuras de aceleración en escenas dinámicas

La Figura 4.1 muestra el proceso que seguiremos para comparar nuestras estructuras. Primero, seleccionamos los *frames* de una escena dinámica, la resolución y una estructura. Para cada uno de los *frames* de la escena utilizamos la estructura para renderizarlos. Este proceso consiste en primero construir la escena y luego recorrer la estructura utilizando los rayos. Luego, calculamos el tiempo que se demora en renderizar cada uno de los *frames*. Este tiempo considera tanto el tiempo de construcción como el tiempo de recorrido de los rayos en la estructura. Con el tiempo recolectado, calculamos la cantidad

de *frames* que es capaz de renderizar en 1 segundo (FPS). Luego, evaluamos el costo *SAH* de la estructura para poder evaluar su calidad frente a las demás estructuras. Finalmente, repetimos este proceso para todas las estructuras en las diferentes escenas para las 3 resoluciones.

Capítulo 5

EXPERIMENTACIÓN

En este capítulo detallamos la experimentación y resultados obtenidos de la comparación entre las diferentes estructuras. En primer lugar se explicará cómo fueron llevados a cabo los experimentos y de qué manera fueron configurados. Después se presentan las escenas que fueron seleccionadas y las características de cada una. A continuación, se muestran los resultados obtenidos de la experimentación. Finalmente, se discuten los resultados enfocándose en las distintas características de las escenas así como en sus posibles aplicaciones.

5.1 Implementación

Para realizar los experimentos utilizamos una laptop con Intel Core I7-9750H 2.69GHz con 6 núcleos, 2 hilos lógicos por núcleo y 24 GB de memoria RAM. Posterior a esto es necesario definir cómo renderizaremos las escenas y de qué manera configuraremos los experimentos para alcanzar los objetivos planteados en la sección.

Para renderizar las escenas utilizamos la versión 1 de la librería madmann91/bvh, la cual nos permite construir escenas con *BVH* y renderizarlas usando *ray tracing*. Esta librería proporciona diversos algoritmos de *BVH* y cada uno de estos *BVH* está referenciado al trabajo de investigación en donde fue presentado. Esto nos permite saber exactamente qué estructura de aceleración estamos usando y así asemejarnos a las comparaciones que realizan otros trabajos para escenas estáticas. Adicionalmente, la construcción y renderización de las escenas está paralelizada con OpenMP utilizando 12 threads. Para mostrar el renderizado final de los *frames* se usa la libreria SDL.

En el caso de la configuración de los experimentos tenemos que definir como enfocaremos la construcción y el recorrido de las estructuras. Al trabajar con escenas dinámicas buscamos alcanzar la mayor cantidad de *frames* para cada escena, por lo que configuraremos los algoritmos en función de este ideal. El algoritmo *LBVH* al ser de los algoritmos más veloces a la hora de construirse será el estándar a alcanzar por parte de los otros dos, refiriéndonos específicamente al tiempo de construcción. En el caso de *PLOC* podemos configurar el rango de búsqueda, a menor rango menor será el tiempo de construcción sacrificando la calidad de la estructura. Elegimos un rango de 5 para tratar de alcanzar el tiempo de construcción de *LBVH*. Finalmente, *KBVH* permite seleccionar el porcentaje de particionamiento que se quiere realizar en toda la escena. De la misma forma, elegiremos un porcentaje de 5 % para disminuir el tiempo de construcción que será el factor más costoso.

Por otro lado, en el caso del recorrido de las estructuras, de los distintos tipos de rayos explicados usaremos el primario. Lanzaremos un rayo por píxel desde la cámara hacia la escena para poder determinar qué objetos son visibles. Se decidió optar por este rayo ya que es el encargado de determinar qué objetos aparecen en pantalla. Además, los rayos primarios son una generalización de cualquier tipo de rayo ya que el recorrido dentro de la estructura vendría a ser el mismo.

5.2 Escenas

El grupo de escenas dinámicas seleccionado proviene de *UNC Dynamic Scene Benchmarks*, el cual es utilizado en trabajos de *ray tracing* [23], [35], así como en diferentes trabajos de detección de colisiones. Este *benchmark* separa cada uno de los *frames* de una escena en archivos distintos en formato PLY, los cuales fueron transformados a archivos OBJ para que puedan ser leídos por la librería. Las 4 escenas de este *benchmark* se encuentran descritas en el Cuadro 5.1.

Escena	Triángulos	Número de *Frames*
Cloth-ball	92k	94
N-body	146k	76
Dragon	252k	16
Lion	1.6M	45

TABLA 5.1: Escenas a evaluar donde se especifica la cantidad de triángulos que estas contienen así como la cantidad de *frames*. La escena menos compleja es la de *Cloth-ball* por tener la menor cantidad de triángulos así como solo representar un tipo de movimiento y mantener la topología de los objetos. Por otro lado, *Lion* es la escena más compleja por tener la mayor cantidad de triángulos y representar una colisión entre 2 objetos y no mantener la topología

No se tomaron en cuenta las escenas de *Letters* y *Funnel*. La primera escena no se consideró porque no se proporciona de la misma forma que las demás otorgando cada *frame* de la escena en formato PLY para luego transformarlo a OBJ. En el caso de *Funnel*, esta escena representa una tela entrando en un embudo el cual es transparente. Cómo evaluamos los experimentos utilizando rayos primarios estos no generan efectos de refracción, por lo que gran parte de la escena no se apreciaría. Por esta razón sólo consideraron las 4 escenas restantes.

El conjunto de escenas contiene diferentes tipos de animaciones que servirán para evaluar el comportamiento de las estructuras en cada uno de los casos. Con respecto a los tipos de movimientos, las escenas *Dragon* y *Lion* tienen un movimiento semi-jerárquico y no mantienen su topología intacta ya que un objeto en descenso colisiona con otro y lo rompe. La escena *N-body* tiene un movimiento jerárquico ya que las esferas o *bodies* se mueven cada una bajo una misma transformación y mantienen su topología. Finalmente, la escena *Cloth-ball* mantiene su topología pero tiene un movimiento incoherente ya que las primitivas del objeto que envuelve la pelota tienen transformaciones diferentes. Los

frames de las escenas se pueden apreciar en las Figuras 5.1 (*Cloth-ball*), 5.2 (*N-body*), 5.3 (*Dragon*) y 5.4 (*Lion*).

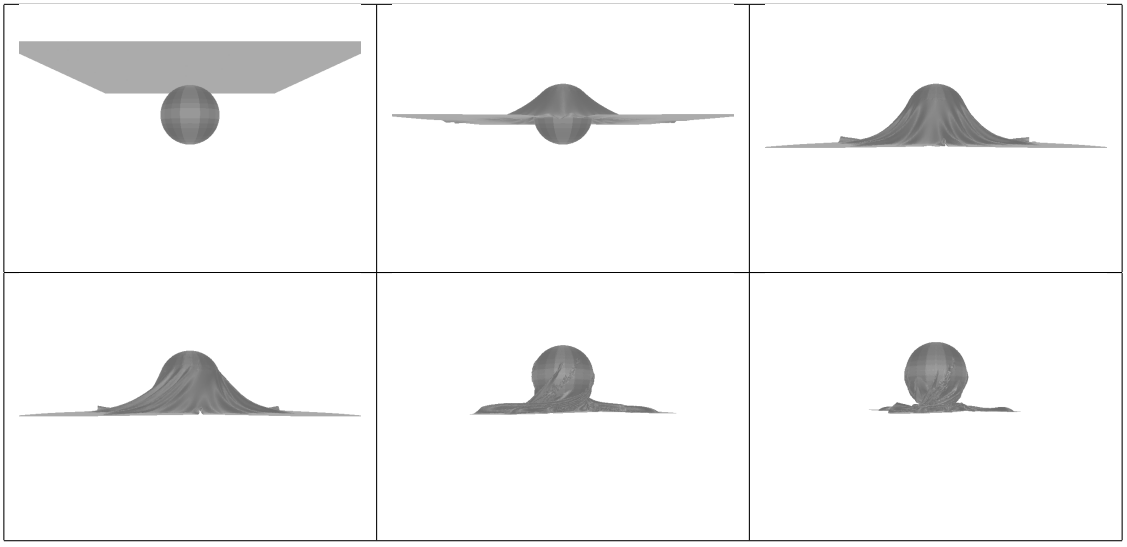

FIGURA 5.1: Representación de algunos *frames* de la escena *Cloth-ball*. Se presentan los *frames* 0, 14, 34, 51, 75 y 93 ordenados de izquierda a derecha y de arriba hacia abajo

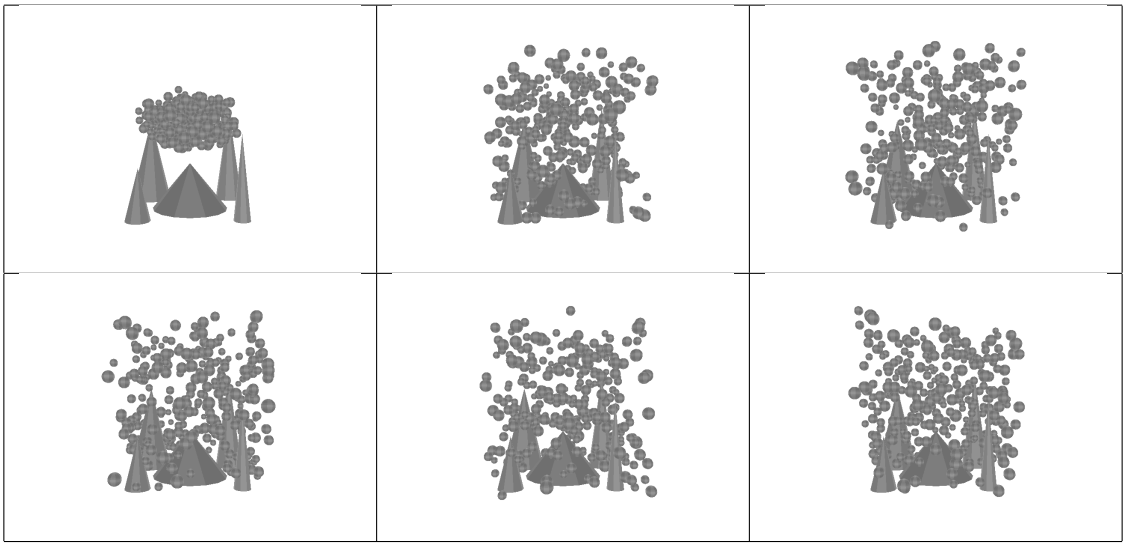

FIGURA 5.2: Representación de algunos *frames* de la escena *N-body*. Se presentan los *frames* 0, 17, 38, 52, 65 y 75 ordenados de izquierda a derecha y de arriba hacia abajo

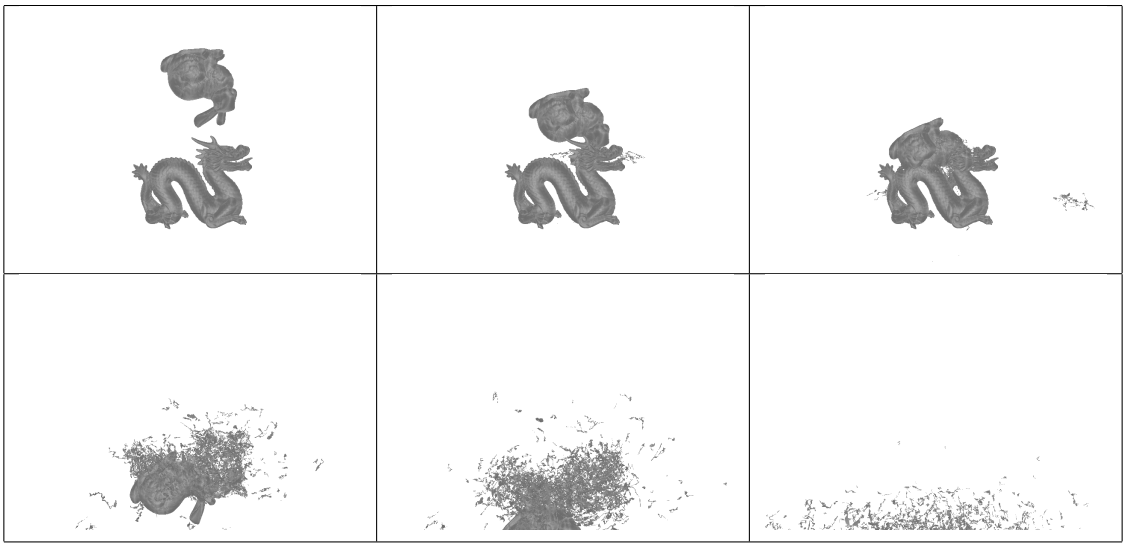

FIGURA 5.3: Representación de algunos *frames* de la escena *Dragon*. Se presentan los *frames* 0, 2, 4, 6, 7 y 12 ordenados de izquierda a derecha y de arriba hacia abajo

5.3 Resultados

Se presentan los resultados en 4 cuadros. Estos están construidos en base a evaluar los *FPS* y la calidad de las estructuras. Los mejores resultados para cada experimento están resaltados en negrita. Para todos los experimentos se realizó un total de 10 repeticiones y se calculó el promedio para determinar los resultados finales.

En primer lugar, en el Cuadro 5.2 se agrupan la cantidad de *FPS* que son capaces de renderizar cada estructura en las 4 escenas. Esta evaluación está separada por las 3 resoluciones propuestas (854x480, 1920x1080 y 3840x2160). Como hemos mencionado a lo largo de este trabajo, para poder calcular los *FPS* es necesario evaluar tanto el tiempo de construcción como el de renderización por lo que ambos resultados han sido desglosados. El tiempo de construcción se encuentra en el Cuadro 5.3 . Estos tiempos no están sujetos a las resoluciones ya que este proceso sólo depende del modelo, por lo que para cada estructura es presentado el tiempo de construcción promedio de todos los *frames* de las 4 escenas. En el caso de la renderización esta si depende de la resolución ya que este

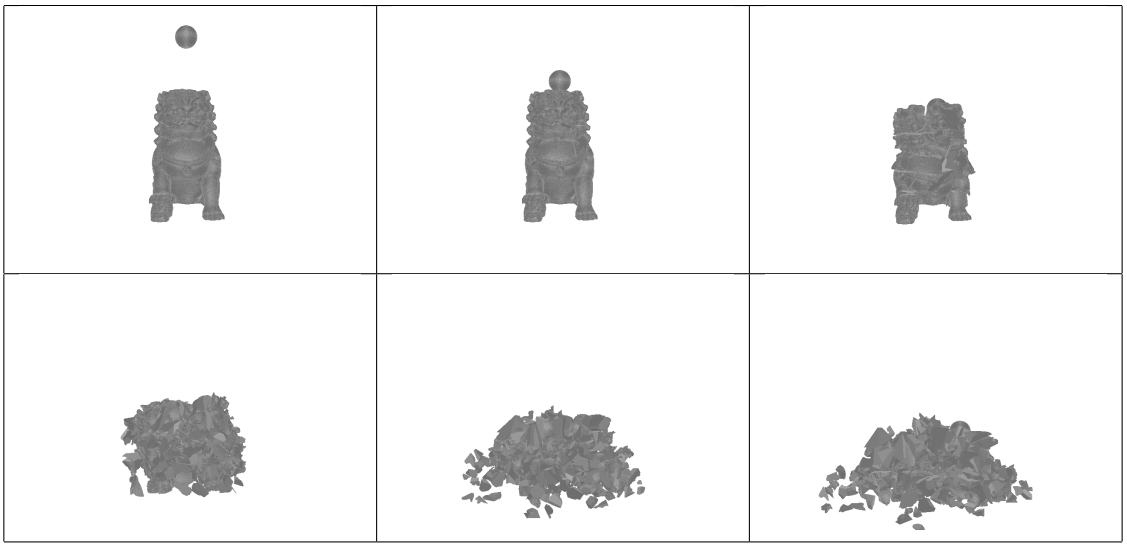

FIGURA 5.4: Representación de algunos *frames* de la escena *Lion*. Se presentan los *frames* 0, 8, 17, 30, 39 y 44 ordenados de izquierda a derecha y de arriba hacia abajo

proceso se basa en el recorrido de los rayos en la estructura y la cantidad de rayos varía según la resolución. Por este motivo, separamos los tiempos de renderización de todos los *frames* en las 3 resoluciones las cuales se encuentran en el Cuadro 5.4.

El Cuadro 5.5 presenta el costo *SAH* para las 4 estructuras en las diferentes escenas. Al explicar esta heurística mencionamos que se utilizan 2 valores conocidos: el costo de recorrido (C_t) y el costo de intersección (C_i). No es muy común que las investigaciones especifiquen los valores de estas constantes. Más importante que los valores exactos es el ratio entre las 2 constantes, los cuales son asignados según cada autor [2]. Normalmente se espera que el costo de recorrido sea mayor que el costo de intersección con una primitiva. Decidimos asignarles los valores de $C_t = 3$ y $C_i = 2$ para realizar nuestros cálculos basándonos en los valores que se usaron en el trabajo donde se utilizó *PLOC* [14][15]. De esta forma mantenemos consistencia con los trabajos del estado del arte.

Finalmente, para facilitar la visualización de los resultados se representan los datos en gráficas. La figura 5.5 presenta los tiempos de construcción promedio para las 4 escenas ordenadas de menor a mayor cantidad de triángulos. Para las 3 resoluciones se

presentaron las figuras 5.6, 5.7 y 5.8 donde se agrupan por escena los tiempos de renderización de cada algoritmo para visualizar como escala esta variable con el aumento de rayos. Finalmente, las gráficas 5.9, 5.10, 5.11 y 5.12 juntan el tiempo de construcción y renderización promedio por cada algoritmo en las 3 resoluciones para visualizar qué parte del proceso impactó más en el cálculo de los *FPS*. Esto se realiza para las 4 escenas por separado.

	Cloth-ball	*N-body*	*Dragon*	*Lion*
FPS (3840x2160)				
LBVH	**0.856**	0.259	0.302	0.092
KBVH	0.75	0.234	0.257	**0.126**
PLOC	0.838	**0.292**	**0.327**	0.097
FPS (1920x1080)				
LBVH	**3.2**	1.002	**1.128**	**0.337**
KBVH	2.016	0.741	0.691	0.218
PLOC	2.733	**1.038**	1.051	0.26
FPS (854x480)				
LBVH	**12.991**	**4.458**	**4.659**	**1.183**
KBVH	3.711	1.798	1.292	0.272
PLOC	7.193	3.28	2.562	0.479

TABLA 5.2: Comparación de los FPS de los distintos algoritmos en las 4 escenas animadas en las 3 resoluciones.

	Cloth-ball	*N-body*	*Dragon*	*Lion*
Tiempo de Construcción Promedio por # de *frames* (ms)				
LBVH	**15**	**25**	**44**	**304**
KBVH	205	339	595	3417
PLOC	76	119	230	1605

TABLA 5.3: Comparación de los tiempos de construcción de los distintos algoritmos en las 4 escenas animadas.

	Cloth-ball	*N-body*	*Dragon*	*Lion*
Tiempo de Renderización Promedio por # de *frames* (ms)(3840x2160)				
LBVH	1152	3833	3265	10586
KBVH	1131	3944	3292	**4548**
PLOC	**1119**	**3312**	**2830**	8777
Tiempo de Renderización Promedio por # de *frames* (ms)(1920x1080)				
LBVH	297	972	842	2665
KBVH	290	1010	850	**1155**
PLOC	**287**	**844**	**720**	2235
Tiempo de Renderización Promedio por # de *frames* (ms)(854x480)				
LBVH	**60**	197	170	542
KBVH	61	212	174	**239**
PLOC	61	**181**	**155**	458

TABLA 5.4: Comparación de los tiempos de renderización de los distintos algoritmos en las 4 escenas animadas en las 3 resoluciones.

	Cloth-ball	*N-body*	*Dragon*	*Lion*
SAH promedio por # de frames				
LBVH	99.72	184.28	83.02	942.98
KBVH	101.62	197.78	83.12	**560.55**
PLOC	**86.56**	**154.75**	**68.25**	724.65

TABLA 5.5: Comparación del costo *SAH* promedio de los distintos algoritmos en las 4 escenas animadas.

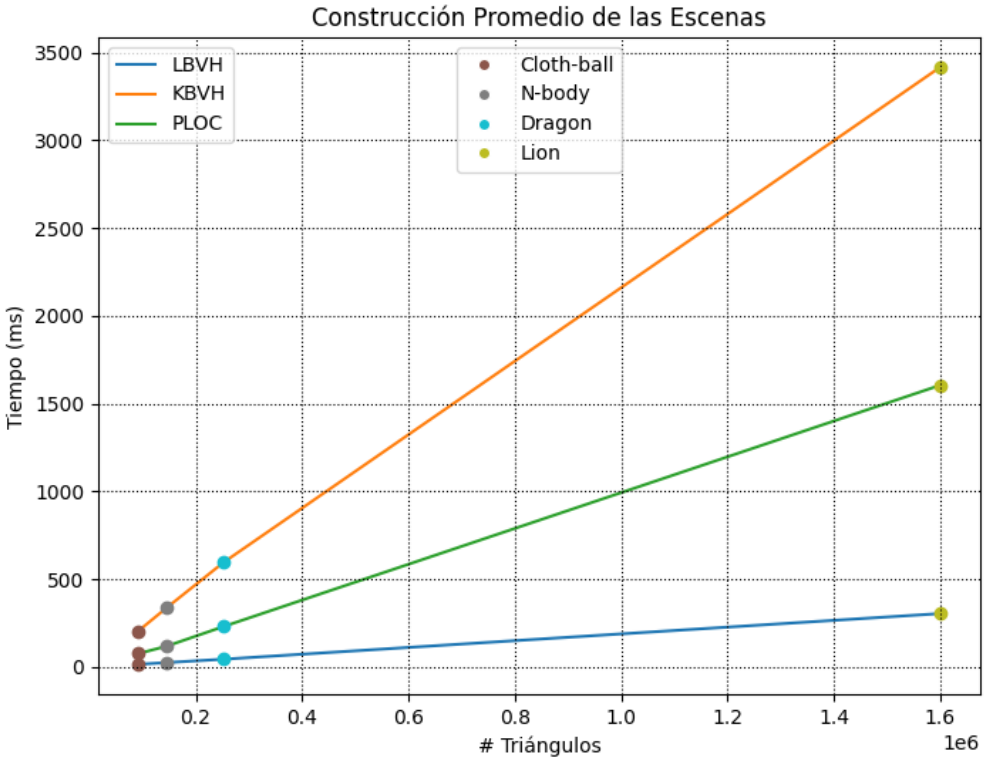

FIGURA 5.5: Tiempo de construcción promedio de los 3 algoritmos según el número de triángulos. Se resalta la posición de las 4 escenas de acuerdo a la cantidad de triángulos que tiene cada una. *KBVH* es el algoritmo que más tiempo consume a la hora de construir las escenas, mientras que *LBVH* es el algoritmo más rápido en la construcción

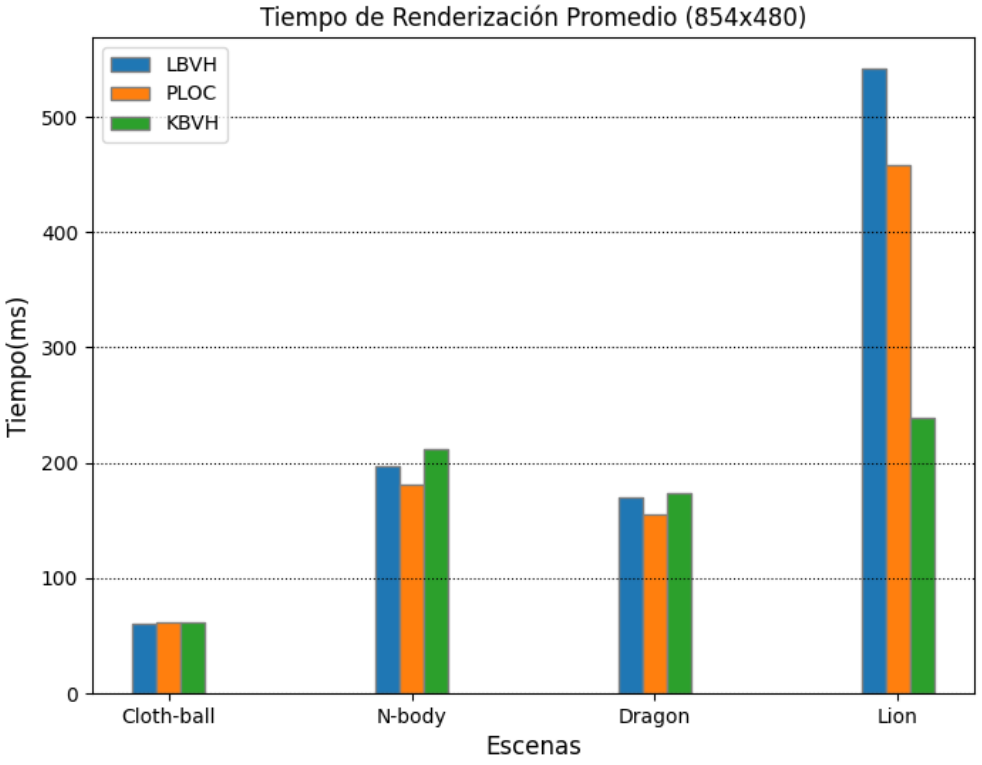

FIGURA 5.6: Tiempo de renderización promedio de los 3 algoritmos agrupados por cada una de las 4 escenas para la resolución 854x480. En la escena *Cloth-ball* los tiempos de renderización son muy similares con una pequeña ventaja por parte de *LBVH*. *PLOC* consigue los mejores tiempos para las escenas *N-body* y *Dragon*. *KBVH* obtiene el mejor tiempo para la escena *Lion*

5.4 Discusión

Interpretaremos los resultados obtenidos para el tiempo de construcción, calidad y renderización, *FPS* y finalmente veremos cómo aplicar este conocimiento a los casos aplicativos de *ray tracing* y escenas dinámicas.

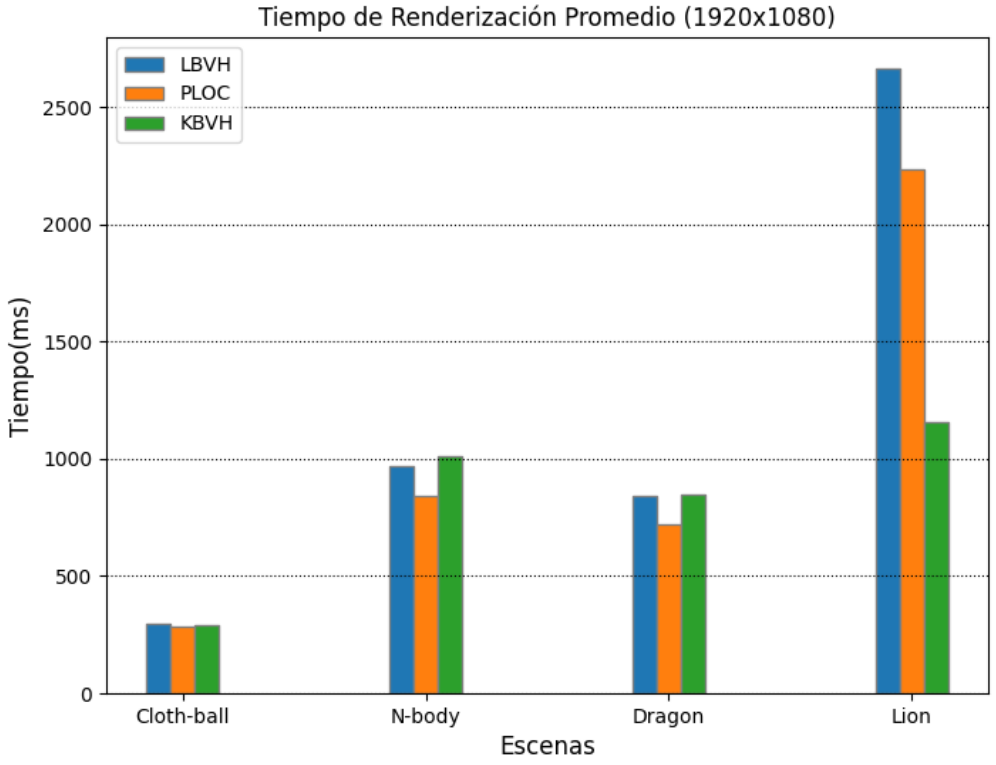

FIGURA 5.7: Tiempo de renderización promedio de los 3 algoritmos agrupados por cada una de las 4 escenas para la resolución 1920x1080. En la escena *Cloth-ball* los tiempos de renderización son muy similares con una pequeña ventaja por parte de *PLOC*. *PLOC* consigue los mejores tiempos para las escenas *N-body* y *Dragon*. *KBVH* obtiene el mejor tiempo para la escena *Lion*

5.4.1 Construcción

Al evaluar el tiempo de construcción en el cuadro 5.3 y en la figura 5.5 podemos ver que para las 4 escenas *LBVH* es el algoritmo que obtiene los mejores resultados. Esto es un resultado esperado, considerando que según la literatura es de las estructuras más veloces a la hora de construir las escenas. Como se explicó en la sección 3.3.2 *LBVH* reduce su construcción a un problema de ordenamiento basado en los Morton Codes y usa eso como base para construir la estructura. Esto no sucede para *PLOC* que a pesar

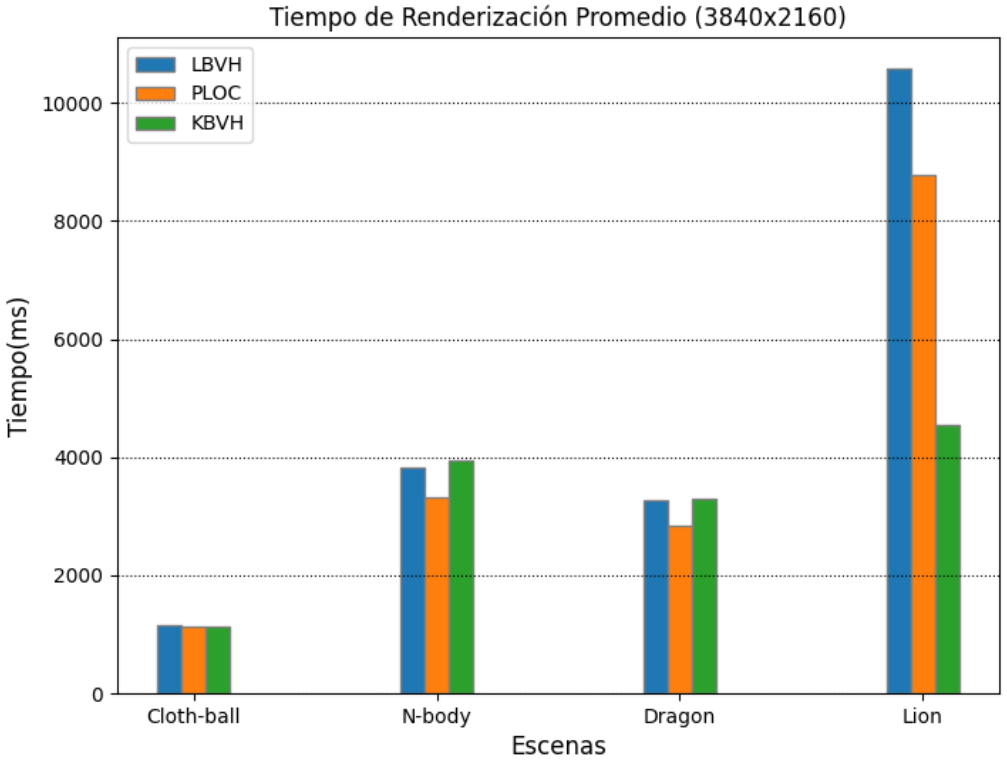

FIGURA 5.8: Tiempo de renderización promedio de los 3 algoritmos agrupados por cada una de las 4 escenas para la resolución 3840x2160. En la escena *Cloth-ball* los tiempos de renderización son muy similares. Sin embargo, en esta resolución *LBVH* tiene el peor tiempo de las 3 estructuras.. *PLOC* consigue los mejores tiempos para las escenas *N-body* y *Dragon*. KBVH obtiene el mejor tiempo para la escena *Lion*

de realizar un ordenamiento basado en Morton Codes también realiza una búsqueda paralela para obtener los vecinos óptimos para cada primitiva, lo cual consume más tiempo. Asimismo, podemos observar como esta ventaja es aún mucho más notoria mientras más grande es la escena, ya que para la escena *Lion*, la diferencia de tiempo de construcción es significativa comparado con el tiempo de *KBVH*. Este factor será determinante a la hora de evaluar los *FPS*, ya que otorga una ventaja considerable a *LBVH* con respecto a los demás. Adicionalmente, se aprecia que el tiempo de construcción crece en función a la cantidad de triángulos de la escena, y esto se cumple para los 3 algoritmos de *BVH*.

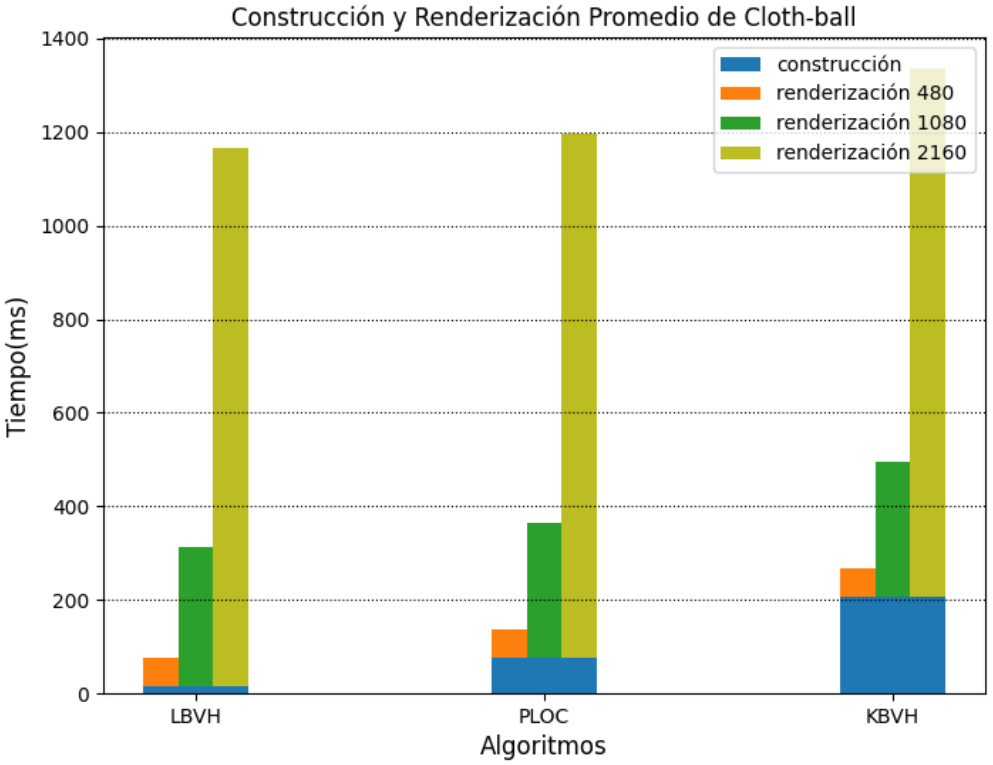

FIGURA 5.9: Agrupación de los tiempos de construcción y renderización promedio por algoritmo en las 3 resoluciones para la escena *Cloth-ball*. *LBVH* consigue el menor tiempo de construcción+renderización promedio para las 3 resoluciones. *KBVH* es el algoritmo que peor tiempo de construcción+renderización promedio tiene

5.4.2 Calidad y Renderización

Evaluamos de manera conjunta la calidad, medida por la heurística *SAH*, con el tiempo de renderización ya que se espera que a mejor calidad, menor sea el tiempo de renderización.

En primer lugar, al evaluar las gráficas 5.6, 5.7 y 5.8 es posible apreciar la similitud que tienen entre ellas, lo que nos hace entender que el tiempo de renderización ha escalado de forma similar para los 3 algoritmos en las 3 resoluciones. Es decir, la variación de rayos en la escena no ha generado casos donde un algoritmo renderice mejor o peor que otro

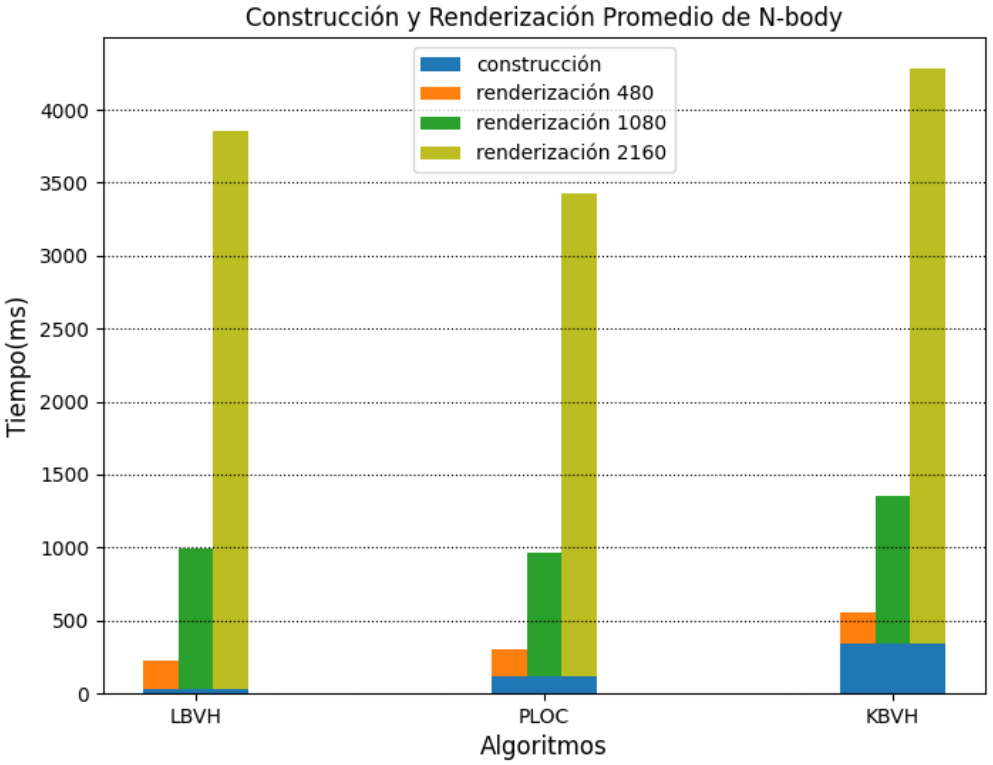

FIGURA 5.10: Agrupación de los tiempos de construcción y renderización promedio por algoritmo en las 3 resoluciones para la escena *N-body*. *LBVH* consigue el menor tiempo de construcción+renderización promedio para la resolución 854x480. *PLOC* tiene el mejor tiempo de construcción+renderización promedio para las resoluciones 1920x1080 y 3840x2160. *KBVH* es el algoritmo que obtiene el peor tiempo de construcción+renderización promedio para las 3 resoluciones

según cambia la resolución. Esto se aprecia de forma clara para todas las escenas, excepto para *Cloth-ball* por lo que hay que analizarlo más a fondo.

Se espera que los algoritmos que tienen mejor calidad para una escena, tengan de la misma forma mejor tiempo de renderización para la misma escena. El cuadro 5.5 nos muestra como *PLOC* obtiene la mejor calidad para *Cloth-ball*, *N-body* y *Dragon*; y *KBVH* para *Lion*. Los resultados que se presentan en el cuadro 5.4 cumplen con esta condición excepto para la escena *Cloth-ball*. Al visualizar las figuras 5.6, 5.7 y 5.8, podemos ver como los tiempos de renderización son prácticamente los mismos para esta escena, cosa

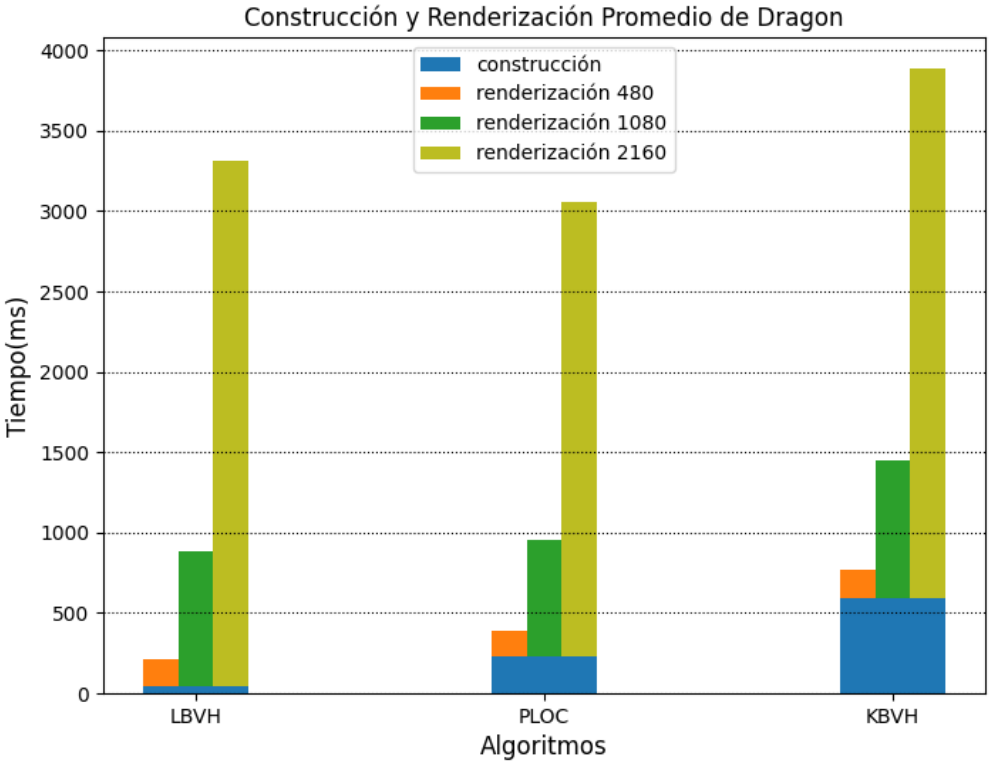

FIGURA 5.11: Agrupación de los tiempos de construcción y renderización promedio por algoritmo en las 3 resoluciones para la escena *Dragon*. *LBVH* consigue el menor tiempo de construcción+renderización promedio para las resoluciones 854x480 y 1920x1080. *PLOC* tiene el mejor tiempo de construcción+renderización promedio para la resolución 3840x2160. *KBVH* es el algoritmo que obtiene el peor tiempo de construcción+renderización promedio para las 3 resoluciones

que no pasa en las demás escenas, donde la relación entre *SAH* y tiempo de renderización se mantiene y con diferencias más marcadas. Si evaluamos los valores de calidad del cuadro 5.5 y la renderización en el cuadro 5.4 vemos que no guardan relación para esta escena. *KBVH* tiene la peor calidad pero para las resoluciones 1920x1080 y 3840x2160 obtiene mejor tiempo que *LBVH* y para la menor resolución los tiempos son prácticamente iguales. Esto se debe a que *Cloth-ball* es la escena con menor número de triángulos, por lo que la calidad *SAH* no influye tanto en el recorrido porque la estructura es más pequeña y si a esto le agregamos el uso de pocos rayos entonces los tiempos terminan siendo muy

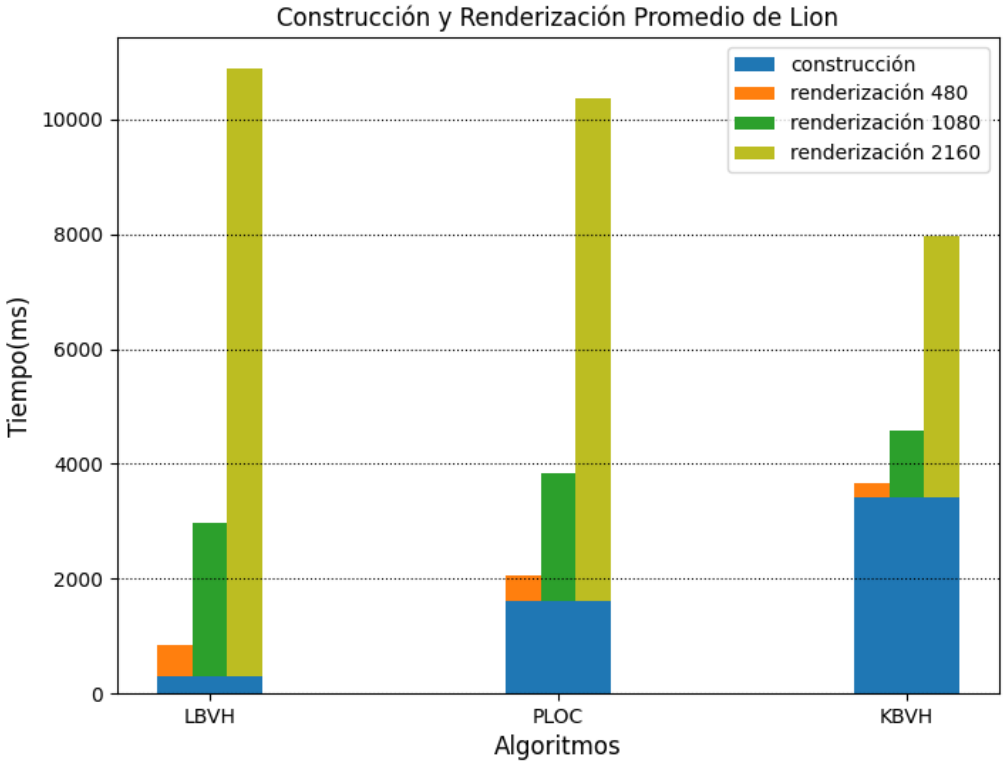

FIGURA 5.12: Agrupación de los tiempos de construcción y renderización promedio por algoritmo en las 3 resoluciones para la escena *Lion*. *LBVH* consigue el menor tiempo de construcción+renderización promedio para las resoluciones 854x480 y 1920x1080 y tiene el peor tiempo construcción+renderización promedio para la resolución 3840x2160. *KBVH* obtiene el mejor tiempo de construcción+renderización promedio para la resolución 3840x2160

similares.

Para terminar de corroborar la idea analizaremos el caso contrario, que sería *Lion*, la escena más grande. *KBVH* tiene la mejor calidad según la heurística *SAH* para esta escena. Si observamos las gráficas 5.6, 5.7 y 5.8 podemos ver como *KBVH* tiene el mejor tiempo de renderización para los 3 casos. Además, comparado con otras escenas, la diferencia de tiempo contra los otros algoritmos es mayor, por lo que se refuerza la idea de que mientras más grande la escena, la calidad de la estructura influirá más en la renderización.

Adicionalmente, al relacionar las características de cada algoritmo, que son descritas en la sección 3.3.2, podemos ver cómo las decisiones tomadas en el diseño de cada estructura se ven reflejadas en nuestros resultados. *LBVH* consigue una veloz construcción al centrarse principalmente en el ordenamiento de las primitivas. Sin embargo, no tiene un criterio extra a la hora de agrupar las primitivas y crear los nodos. Esto puede generar un agrupamiento subóptimo de las primitivas, el cual será mucho más notorio según aumente la cantidad de primitivas ya que es más probable juntar primitivas que no necesariamente son las más cercanas entre sí. Esto se puede ver en nuestros resultados al observar cómo obtiene el peor tiempo para las 3 resoluciones en la escena más grande. Por otro lado, *PLOC* a pesar de utilizar el ordenamiento de las primitivas como base de su construcción agrega una fase de búsqueda del vecino más cercano para tratar de optimizar la creación de nodos y obtener una mejor calidad. Esto le ocasiona tener una construcción más lenta que *LBVH* pero le permite tener una mejor calidad ya que en ninguna de las escenas obtuvo el peor tiempo de renderización y en las escenas *N-body* y *Dragon* obtiene los mejores tiempos para las 3 resoluciones. Finalmente, *KBVH* obtiene la mejor calidad y con diferencia en la escena *Lion* la cual es la más grande. Esto se debe a que al trabajar sobre una escena con mayor número de primitivas, optimizará una mayor cantidad de ellas al particionarlas antes de realizar la construcción, obteniendo una mejor calidad que se nota después en el tiempo de renderización.

5.4.3 *FPS* por escena

Al observar los resultados del cuadro 5.2, podemos ver que solo en la escena Clothball un solo algoritmo ha conseguido la mayor cantidad de *FPS*, en este caso *LBVH*. Esto se debe principalmente al tiempo de construcción. Al observar la gráfica 5.9, vemos como los tiempos de renderización para los 3 algoritmos son muy similares en tamaño, por lo que esa ventaja en la construcción le da a *LBVH* esa ventaja en *FPS* para la escena más pequeña. Bajo este mismo concepto, mientras menos influya el tiempo de renderización,

más útil será la estructura *LBVH*, por lo que para todas las escenas en la menor resolución, o donde haya una poca cantidad de rayos disparados, este algoritmo es el que consigue los mejores resultados de *FPS*, como se observa en el cuadro 5.2.

En el caso de la escena *N-body*, dejando de lado la resolución menor, podemos ver como *PLOC* consigue el mejor resultado en *FPS*. Como podemos observar en la figura 5.10, esta ventaja se consigue principalmente en el tiempo de renderización y esa ventaja es aún más marcada en la resolución mayor, es decir mejora con el aumento en la cantidad de rayos. Esta escena tiene un movimiento jerárquico y mantiene su topología, por lo que *PLOC* parece una buena opción para estos casos.

Para corroborar esta idea observamos los resultados de la escena *Dragon*. Según los resultados del cuadro 5.2, *PLOC* obtiene el mejor resultado para la resolución mayor y *LBVH* para las otras 2 resoluciones. Sin embargo, en la resolución 1920x1080, la gráfica 5.11 muestra que la diferencia entre *PLOC* y *LBVH* es mínima. Esta escena si bien tiene un movimiento semi-jerárquico y no mantiene su topología, gran parte de la escena es el conejo que desciende y colisiona con el dragón. Este objeto sigue un movimiento jerárquico y mantiene su topología, similar al caso de *N-body*. Por lo tanto, vemos como *PLOC* funciona de manera efectiva para este tipo de escenas que tienen objetos que mantienen sus topología con un movimiento jerárquico.

En el caso de la escena *Lion*, esta guarda cierta similitud con la escena *Dragon*, ya que un objeto que sigue un movimiento jerárquico y manteniendo su topología entra en colisión con otro al cual rompe. En esta escena, *LBVH* consigue los mejores resultados en *FPS* para las resoluciones de 854x480 y 1920x1080. Sin embargo, para la resolución de 3840x2160 *LBVH* pasa a ser el peor algoritmo y *KBVH* obtiene la mejor cantidad de *FPS*. Si bien, esta escena es similar a *Dragon*, consideramos que la mayor diferencia es el tamaño de los objetos que colisionan con el objeto a romper. En el caso de *Dragon*, el objeto era grande y ocupaba gran parte de la escena, pero para *Lion*, el objeto es pequeño y se pierde entre los escombros del objeto al romperse. Sin embargo, *PLOC* consigue superar

a *LBVH* en la mayor resolución, por lo que guarda cierta relación con los resultados de la escena anterior. *KBVH* obtiene los mejores *FPS* en la resolución mayor y esto se debe principalmente a su tiempo de renderización que explicamos en la sección anterior. Por lo que concluimos que *KBVH* funciona bien para escenas con movimientos semi-jerárquicos pero que principalmente sean muy grandes y que se utilicen una gran cantidad de rayos.

5.4.4 Aplicaciones

Como se menciona en la sección 1.1, una de las aplicaciones de *ray tracing* en escenas dinámicas es en modelamiento molecular, donde se aprecian los átomos y sus conexiones que forman las moléculas. Para este tipo de casos consideramos que la mejor opción es utilizar *PLOC*. Para esto nos basamos en la escena de *N-body* que se puede asemejar a una representación molecular, ya que representan una gran cantidad de objetos que vendrían a ser los átomos, los cuales mantienen su topología e interactúan entre sí con movimientos jerárquicos.

Otro caso de uso muy común es en los videojuegos. En este ámbito las opciones son más variadas. Uno de los casos más comunes vendrían a ser escenas donde hay explosiones. Para este tipo de escenas se considera que la mejor opción es utilizar un *LBVH*. Consideramos esto principalmente porque es una escena donde existe principalmente un movimiento incoherente, además se busca que sea muy rápido e instantáneo por lo que no se requiere que la calidad sea muy alta. Debido a esto, la cantidad de rayos a disparar podría ser menor, y como hemos visto en nuestros resultados, mientras menor cantidad de rayos son usados *LBVH* obtiene ventaja por su rápida construcción.

Para el caso del entorno, es decir las construcciones, edificios o monumentos de la escena, consideramos que la mejor opción es utilizar un *KBVH*. Como hemos visto, mientras más grande la escena, más influirá el tiempo de renderización, por lo que nos basamos en los resultados obtenidos en la escena *Lion*. Además al ser los objetos más

grandes podrán generar una mayor cantidad de efectos en la escena, por lo que será necesario lanzar una mayor cantidad de rayos. Por otro lado, si estos objetos pueden llegar a ser destruidos y romper su topología hemos visto cómo para este caso *KBVH* responde mejor que las demás estructuras.

Por otro lado, para objetos que mantienen su topología y realizan un movimiento jerárquico, como aviones, automóviles y hasta los propios personajes, la mejor opción vendría a ser *PLOC*. Como hemos visto en los resultados, *PLOC* se desempeña mejor que los demás algoritmos para escenas con un movimiento jerárquico que mantienen su topología. Además, a pesar de que la topología se rompa al ser objetos no tan grandes *PLOC* puede alcanzar una mejor cantidad de *FPS* si es que se utilizan una gran cantidad rayos, como hemos visto en la escena *Dragon*.

Finalmente, el seleccionar una estructura sobre otra va a depender del uso que se le quiera dar a la escena. Esto implica tener en cuenta las diferentes variables que hemos mencionado a lo largo de este trabajo como: tamaño de la escena, cantidad de rayos a utilizar, tipos de movimientos y si se mantiene la topología o no. De esta manera se podrá determinar qué estructura usar, ya que como hemos visto, ninguna estructura es perfecta para todos los casos.

CONCLUSIONES Y TRABAJOS FUTUROS

Ray tracing es una técnica de renderización que simula el comportamiento de la luz. Sin embargo, es muy costosa ya que se deben evaluar todos los rayos con todas las primitivas por lo que el uso de estructuras de aceleración es fundamental en cualquier escena. Las escenas dinámicas tienen una dificultad extra ya que requieren de una construcción y actualización rápida de las estructuras para poder adaptarse a los cambios en la geometría de los objetos de la escena. Sin embargo determinar qué estructura es más óptima no es posible ya que los trabajos comparativos no han evaluado este aspecto.

En este trabajo hemos comparado 3 estructuras de aceleración del estado del arte (*LBVH, KBVH y PLOC*) en un *benchmark* de escenas dinámicas (UNC) para evaluar su desempeño. De esta experimentación hemos recopilado los datos con respecto a la cantidad de *frames* por segundo que son capaces de generar y la calidad de las estructuras en base a la heurística *SAH*. Adicionalmente, hemos realizado nuestros experimentos en 3 resoluciones distintas para observar cómo influye la cantidad de rayos en el desempeño de los algoritmos.

De los resultados obtenidos pudimos observar que *LBVH* tiene un buen desempeño en escenas de menor tamaño y si la resolución es menor (menor número de rayos). Sin embargo, su baja calidad le hace bajar su desempeño según las escenas sean más grandes y complejas y la cantidad de rayos aumenten. Adicionalmente, es útil para escenas donde la topología se rompe siempre y cuando no se use una gran cantidad de rayos. Por otro

lado, *PLOC* obtiene buenos resultados en escenas con muchos objetos de movimiento jerárquico y que mantengan su topología, principalmente por su buen desempeño en la escena *N-body*. Además, se desempeña bien para todas las escenas en la resolución más grande, obteniendo la mayor cantidad de *FPS* en 2 de las 4 escenas y estando cerca de la mejor estructura en las otras 2. Finalmente, *KBVH* no obtuvo los mejores resultados en todas las escenas excepto en la escena más grande (*Lion*) en la mayor resolución.

Para continuar con la investigación de este trabajo una opción es explotar aún más la gran capacidad de paralelismo de los 3 algoritmos. Para esto se podría realizar los experimentos utilizando GPU y obtener resultados con el hardware de última generación. Por otro lado, experimentar con algoritmos que no solo se reconstruyan, sino se actualicen en base a los cambios de la escena para observar otras propuestas del estado del arte. Finalmente ampliar la experimentación al utilizar otro grupo de escenas dinámicas, como *BART* [36] o escenas conocidas como *Fairy Forest 2*. De esta manera corroborar nuestras conclusiones o encontrar nuevos patrones para ampliar el análisis.

REFERENCIAS BIBLIOGRÁFICAS

[1] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the art in ray tracing animated scenes," in *Computer graphics forum*, vol. 28, no. 6. Wiley Online Library, 2009, pp. 1691–1722.

[2] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A Survey on Bounding Volume Hierarchies for Ray Tracing," *Computer Graphics Forum*, 2021.

[3] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 37–45.

[4] J. Burgess, "Rtx on—the nvidia turing gpu," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020.

[5] T. Ize, *Efficient acceleration structures for ray tracing static and dynamic scenes*. University of Utah, 2009.

[6] N. Thrane, L. O. Simonsen *et al.*, "A comparison of acceleration structures for gpu assisted ray tracing," 2005.

[7] F. Ge, "Comparing acceleration data structures for real-time ray tracing on gpu," 2013.

[8] M. Vinkler, V. Havran, and J. Bittner, "Bounding volume hierarchies versus kd-trees on contemporary many-core architectures," in *Proceedings of the 30th Spring Conference on Computer Graphics*, 2014, pp. 29–36.

[9] T. L. Kay and J. T. Kajiya, “Ray tracing complex scenes,” *ACM SIGGRAPH computer graphics*, vol. 20, no. 4, pp. 269–278, 1986.

[10] J. Goldsmith and J. Salmon, “Automatic creation of object hierarchies for ray tracing,” *IEEE Computer Graphics and Applications*, vol. 7, no. 5, pp. 14–20, 1987.

[11] T. Foley and J. Sugerman, “Kd-tree acceleration structures for a gpu raytracer,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005, pp. 15–22.

[12] D. V. Macedo and M. A. F. Rodrigues, “Comparison of acceleration data structures for high quality fast reflections of static and deformable models in walkthrough animations,” *SBC Journal on Interactive Systems*, vol. 7, no. 1, pp. 28–37, 2016.

[13] Z. Li, Y. Deng, and M. Gu, “Path compression kd-trees with multi-layer parallel construction a case study on ray tracing,” in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2017, pp. 1–8.

[14] D. Meister and J. Bittner, “Parallel locally-ordered clustering for bounding volume hierarchy construction,” *IEEE transactions on visualization and computer graphics*, vol. 24, no. 3, pp. 1345–1353, 2017.

[15] ——, “Parallel reinsertion for bounding volume hierarchy optimization,” in *Computer Graphics Forum*, vol. 37, no. 2. Wiley Online Library, 2018, pp. 463–473.

[16] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, “Fast bvh construction on gpus,” in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

[17] I. Wald, “On fast construction of sah-based bounding volume hierarchies,” in *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2007, pp. 33–40.

[18] T. Karras, “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees,” in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, 2012, pp. 33–37.

[19] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, “Efficient bvh construction via approximate agglomerative clustering,” in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 81–88.

[20] P. Ganestam, R. Barringer, M. Doggett, and T. Akenine-Möller, “Bonsai: rapid bounding volume hierarchy generation using mini trees,” *Journal of Computer Graphics Techniques*, vol. 4, no. 3, pp. 23–42, 2015.

[21] T. Karras and T. Aila, “Fast parallel construction of high-quality bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 89–99.

[22] C. Benthin, S. Woop, I. Wald, and A. T. Áfra, “Improved two-level bvhs using partial re-braiding,” in *Proceedings of High Performance Graphics*, 2017, pp. 1–8.

[23] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, “Fast, effective bvh updates for animated scenes,” in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2012, pp. 197–204.

[24] I. Wald, C. Benthin, and P. Slusallek, “Distributed interactive ray tracing of dynamic scenes,” in *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.* IEEE, 2003, pp. 77–85.

[25] T. Aila, T. Karras, and S. Laine, “On quality metrics of bounding volume hierarchies,” in *Proceedings of the 5th High-Performance Graphics Conference*, 2013, pp. 101–107.

[26] T. Whitted, “An improved illumination model for shaded display,” in *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, 1980, p. 14.

[27] E. Haines and T. Akenine-Möller, *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, 2019.

[28] R. L. Cook, T. Porter, and L. Carpenter, “Distributed ray tracing,” in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984, pp. 137–145.

[29] T. Möller and B. Trumbore, “Fast, minimum storage ray-triangle intersection,” *Journal of graphics tools*, vol. 2, no. 1, pp. 21–28, 1997.

[30] P. Schneider and D. H. Eberly, *Geometric tools for computer graphics*. Elsevier, 2002.

[31] B. Smits, “Efficient bounding box intersection,” *Ray tracing news*, vol. 15, no. 1, 2002.

[32] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, “An efficient and robust ray-box intersection algorithm,” in *ACM SIGGRAPH 2005 Courses*, 2005, pp. 9–es.

[33] N. Sousa, D. Sena, N. Papadopoulos, and J. Pereira, “Acceleration data structures for ray tracing on mobile devices.” in *VISIGRAPP (1: GRAPP)*, 2019, pp. 332–339.

[34] I. Wald and V. Havran, “On building fast kd-trees for ray tracing, and on doing that in o (n log n),” in *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2006, pp. 61–69.

[35] S.-E. Yoon, S. Curtis, and D. Manocha, “Ray tracing dynamic scenes using selective restructuring,” in *Eurographics Symposium on Rendering 2007 (EGSR 2007)*. EGSR, 2007.

[36] J. Lext, U. Assarsson, and T. Moller, “A benchmark for animated ray tracing,” *IEEE Computer Graphics and Applications*, vol. 21, no. 2, pp. 22–31, 2001.