An evaluation of Kd-Trees vs Bounding Volume Hierarchy (BVH) acceleration structures in modern CPU architectures

Evaluación de estructuras de aceleración Kd-Trees vs BVH en arquitecturas CPU modernas

Ernesto Rivera-Alvarado[1], Julio Zamora-Madrigal[2]

Fecha de recepción: 25 de enero, 2022
Fecha de aprobación: 16 de mayo, 2022

Rivera-Alvarado, E; Zamora-Madrigal, J. An evaluation of Kd-Trees vs Bounding Volume Hierarchy (BVH) acceleration structures in modern CPU architectures. *Tecnología en Marcha*. Vol. 36, Nº 2. Abril-Junio, 2023. Pág. 86-98.

https://doi.org/10.18845/tm.v36i2.6098

1 Instituto Tecnológico de Costa Rica. Costa Rica.
Correo electrónico: errivera@itcr.ac.cr
https://orcid.org/0000-0002-2756-0572
2 Instituto Tecnológico de Costa Rica. Costa Rica.
Correo electrónico: juzamora95@gmail.com
https://orcid.org/0000-0001-5135-2717

Keywords

Ray tracing; CPU; acceleration structures; modern hardware; BVH; Kd-Trees, bounding volume hierarchy.

Abstract

Ray tracing is a rendering technique that is highly praised for its realism and image quality. Nonetheless, this is a computationally intensive task that is slow compared to other rendering techniques like rasterization. Bounding Volume Hierarchy (BVH) is a primitive subdivision acceleration mechanism that is the mainly used method for accelerating ray tracing in modern solutions. It is regarded as having better performance against other acceleration methods. Another well-known technique is Kd-Trees that uses binary space partitioning to adaptively subdivide space with planes. In this research, we made an up-to-date evaluation of both acceleration structures, using state-of-the-art BVH and Kd-Trees algorithms implemented in C, and found out that the Kd-Trees acceleration structure provided better performance in all defined scenarios on a modern x86 CPU architecture.

Palabras clave

Ray tracing; CPU; estructuras de aceleración; hardware moderno; BVH; Kd-Trees, jerarquía de volúmenes limítrofes.

Resumen

Ray-tracing es una técnica de sintetizado de imágenes que destaca por el gran realismo y calidad que puede brindar en una imagen. Sin embargo, esta técnica es computacionalmente intensiva y lenta en comparación a otras metodologías de sintetizado tal como rasterización. Las Jerarquías de Volúmenes Limítrofes (BVH por sus siglas en inglés) son un mecanismo de aceleración basado en subdivisión de primitivas cuyo principal uso es la aceleración de ray-tracing en soluciones modernas. Comúnmente es referida como una solución que provee mejor aceleración con respecto a otras técnicas. Otra técnica bien conocida son los árboles-Kd (Kd-Trees en inglés), los cuales usan particionamiento de espacio binario con división adaptativa de planos. En esta investigación realizamos una evaluación actualizada de ambas estructuras de aceleración, usando algoritmos del estado del arte de BVH y Kd-Trees implementados en el lenguaje C. Entre los hallazgos encontramos que Kd-Trees provee mejor aceleración en una arquitectura x86 moderna para los escenarios planteados.

Introduction

Ray tracing is the holy grail in computer graphics when it comes to its capacity of rendering realistic, high-quality images [1], but this is paid with rendering time, ray tracing is considered to be slow when compared against other techniques such as rasterization [1].

The way that ray tracing works is by defining a tridimensional scene that contains an origin (represented by a *(x, y, z)* coordinate), a projection plane, and several objects (primitives) that are part of the scene [2]. Mathematical rays are cast from the origin to each one of the pixels from a projection plane that represents the screen image that a user sees. If an intersection is detected between the ray and one or several objects from the scene, the color of the object with the closest intersection to the origin is the one that is painted for that pixel. This process repeats for all the pixels that are part of the projection plane. [2]

The algorithm for a basic ray tracer is quite simple. It is composed of a nested loop in which the external loop iterates through the vertical pixels of the projection plane, and the internal loop iterates through the horizontal pixels [9]. A ray is generated for each coordinate (composed of the horizontal pixel *x* and the vertical pixel *y*) to calculate its intersection (if any) with the nearest object from the scene, in order to print the color of that object in the screen. If no intersection is detected, a default color is assigned [2]. Figure 1 displays the pseudocode of a ray tracer.

```
Algorithm I.1: RayTracer($Scene$)

for $y \leftarrow 1$ to $Scene.VerticalResolution$
  do
    for $x \leftarrow 1$ to $Scene.HorizontalResolution$
      do
        $R \leftarrow$ GetRay($x, y, Scene.eye$)
        $I \leftarrow$ GetNearestIntersection($R, Scene$)
        PaintScreen($x, y, I$)
```

Figure 1. Ray tracer pseudocode.

It's well documented that when rendering an image using the ray tracing algorithm, most of the time is spent in computing ray/object intersection [15]. The basic ray tracing technique (without acceleration structures) has an *O(I n)* time complexity. *I* represents the total number of pixels in the image, and *n* is the total number of objects that comprise the scene. Rendering time becomes unmanageable with scenes with a few thousand objects [2].

Several approaches have been developed to improve the ray/object intersection computation time. The two most popular techniques are *Kd-Trees* and *Bounding Volume Hierarchy* (BVH) [15].

The Bounding Volume Hierarchy acceleration structure uses a subdivision of the primitives (objects) that are part of the scene by creating a partitioned hierarchy of disjoint sets. Figure 2 shows the BVH mechanism; the objects of the scene are included in containers to create a binary tree data structure, where the primitives of the scene are in the leaves of the tree, and each intermediate node represents a container that stores the elements of the nodes beneath it [1].

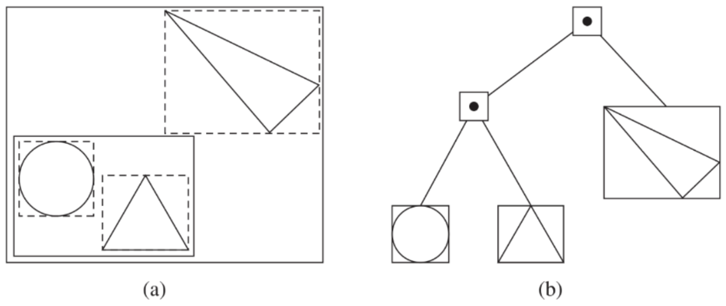

Figure 2. Bounding Volume Hierarchy diagram. Source: [15]

BVH provides rendering acceleration because when a ray is generated, its intersection is calculated for each of the nodes of the tree. If there is not an intersection with a node, all the subtree for that node can be ignored [15]. The performance gain goes from *O(I n)* of the basic ray tracing algorithm to an average of *O(I log n)*, which is the height of the binary tree [15].

Kd-Trees are a variation of *binary space partitioning trees* (BSP) that subdivides tridimensional space with planes. The way that a BSP works is by creating a bounding box that contains the entire scene. If the number of objects included in that box is greater than a defined threshold, the box is split by a plane. This process continues recursively until a maximum defined depth of the tree is reached, or each leaf of the resulting tree contains a sufficiently small number of objects [15]. The splitting planes can be placed in arbitrary positions inside a bounding box as long it is perpendicular to one of the coordinate axes. This property of the Kd-Trees makes their construction and traversal more efficient compared to other types of BSPs [15]. The asymptotic complexity of Kd-Trees is the same as BVH, with an average of *O(I log n)* [15].

Figure 3 shows the Kd-Trees mechanism; the tree is built by recursively splitting the bounding box of the scene geometry perpendicular to one of the coordinate axes. As the figure shows, the first split goes along the *x* axis, so the triangle will be inside the container in the right. The left region is split several more times with axis-aligned planes. [1].

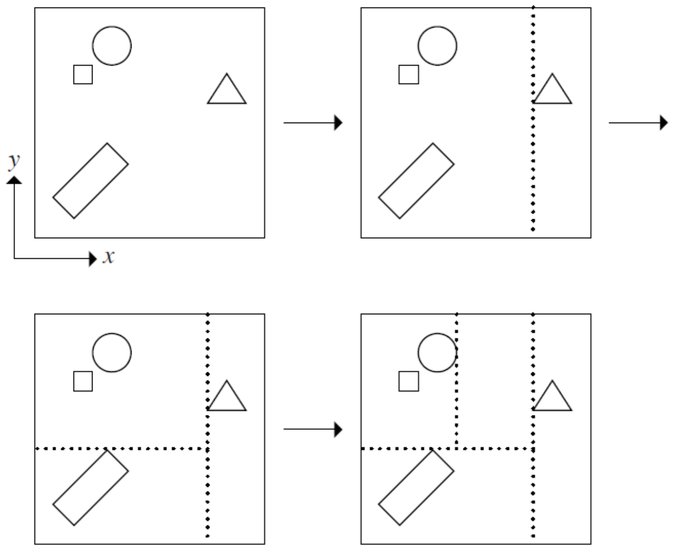

Figure 3. Kd-Trees diagram. Source: [15]

In this research, we evaluated the performance of state-of-the-art BVH and Kd-Trees acceleration structures for ray tracing in a modern CPU architecture. We implemented both methods in the C programming language and explored both algorithm's performance in different scenes that simulate real rendering scenarios. Maintaining up-to-date performance evaluation of the acceleration structures in modern x86 CPUs could lead to new findings regarding its performance. These processors have several internal improvements that could favor a specific acceleration technique. We found that contrary to popular belief, *Kd-Trees* provided better performance than BVH in the defined experiments.

Different efforts for improving the performance of the ray tracing algorithm can be found in Background section. The description of our implementation of the acceleration structures are in Design. Methodology has the hardware used, the experiments and the method of analysis of the obtained data. Results section has the obtained results which are discussed in Discussion. Finally, in the Conclusion and Future Work section we provide our final remarks of the results obtained and propose other problems that could be explored.

Background

There are several different approaches to improve the performance of the ray tracing algorithm. Some works have developed hybrid algorithms that combine ray tracing with rasterization, so for effects that require a lot of computational power, the cheaper rasterization technique is used [9]. There are commercial solutions that used this approach, like Nvidia RTX [6]. The downside is that it is not a pure ray tracing solution, and there is a compromise between image quality (photorealistic effects) and performance.

When using the GPU architecture for rendering an image with ray tracing, RAM-VRAM memory transactions are commonplace, partly due to the relatively small cache of GPU architecture. This makes memory bandwidth a bottleneck for performance [7,15]. Another initiative focused on the reduction of memory transactions to the GPU by compressing the transferred information [21]. This approach reduces the bottleneck caused by the memory bandwidth but adds time due to the compression and decompression of the data in all memory transactions.

As mentioned before, ray/object intersection is where most rendering time is spent [15]. One of the first practical acceleration structure implementation through bounding objects was developed for the CPU architecture by [11]. Since that implementation, several works have been done to improve the performance of the BVH accelerator, like the explicit caching of node-pairs to enhance the access time to the Bounding Volume Test Tree [5]. Another approach has created private workstack in to reduce memory access and inter-thread synchronization. The modifications mentioned above to the BVH algorithm creates new problems such as work-flow divergence and load-imbalance in several cases, which leads to degraded performance [3].

There is research to improve the performance of the BVH acceleration structure through tightly coupled heterogeneous computing [18]. The focus of this research was to utilize the full resources of processors that have an integrated GPU. The performance gain provided by the study was promising. Still, they don't evaluate the performance of the Kd-Trees acceleration structures and require the use of CPUs with integrated GPUs for the acceleration gain.

Another approach for accelerating ray/object intersection are binary partitioning trees (BSP trees), in which the axis-aligned BSP Trees (Kd-Trees) are the most popular for ray tracing [1]. The first reported use of using a Kd-Trees for ray tracing was by [10]. Work to improve the shortcomings of a naive implementation of the Kd-Trees have been done by [4], where it implements a space-efficient representation that avoids redundant objects from leaf nodes. This approach highly improves the performance of the traversal during ray/object intersections.

Performance comparison between BVH and Kd-Trees has been performed in several research efforts, like the one in [19]. This effort is focused only on the GPU architecture, so they don't provide insight into how both algorithms perform on the modern CPUs. Efforts like the one in [12] provides a performance evaluation of both algorithms, where Kd-Trees provided performance better in some scenarios. The main disadvantage of this research is that it was performed a long time ago when the typical CPU architecture only had one core and a very small cache memory [8].

As far as we know, there haven't recent efforts to evaluate the performance of the BVH and Kd-Trees acceleration structures in recent (2018+) mainstream CPU architectures. In this research, we provided an up-to-date comparison of the performance of both algorithms, using our own implementation of state-of-art BVH and Kd-Trees algorithms in the C programming language.

Design

In this research, we created our own ray tracer with BVH and Kd-Trees acceleration heavily based in the state of the art algorithms found in [15] and [1]. We used the C programming language as allow us to implement low-level optimizations for memory access and mathematical operations [22].

For instance, we avoided using pointers for the binary tree data structure of both acceleration methods. Instead, we flattened the tree and stored it in an array. Each element of the array is a struct that represents a node. The first child of the node will be stored in the index next to that node in the array; the second child's index is stored in the struct [15].

Figure 4 shows a pointer representation of a binary tree, while figure 5 shows a flattened representation of the same tree stored in an array, like what we used. The traversal of an array is several times faster than the traversal of the same data structure using pointers [10, 15].

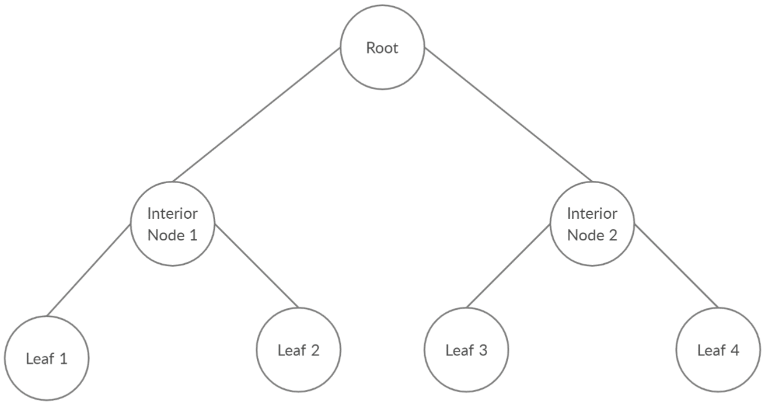

Figure 4. Pointer representation of a binary tree.

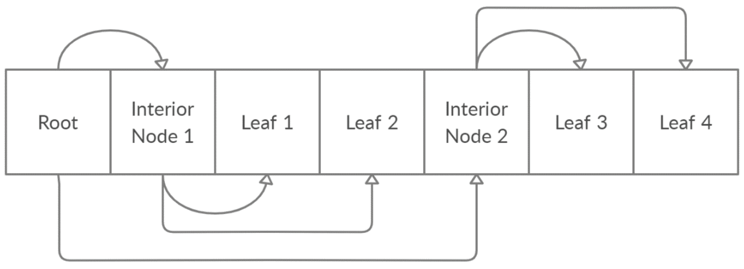

Figure 5. Flattened representation of a binary tree in an array.

To make an evaluation that contemplates real rendering scenarios, we implemented several types of primitives in the ray tracer: The included primitives are:

- Spheres.
- Cones.
- Cylinders.
- Discs.
- Triangles.

Also, we implemented transparencies, reflection, and anti-aliasing effects as they are usually present in real-world use of ray tracing for rendering [1]. We also wanted to evaluate the performance of both algorithms with the presence of both effects.

Methodology

For the evaluation, we choose to employ a Factorial Analysis of Variance (ANOVA) experiment, as it provides a mechanism for evaluating the impact that several factors under research have on a response variable [13]. Using this methodology allowed us the assess the performance of the BVH and Kd-Trees acceleration structures and the effect that other factors had in rendering time.

A factor has different levels that could impact in some way the response variable [14]. ANOVA lets us determine if the influence of a specific factor is statistically significant.

The factors and the levels used for our evaluation are the following:

Objects: The number of objects in the scene directly impacts rendering time. The following amounts were used:

- 1000.
- 4000.
- 7000.
- 10000.
- 13000.

Image resolution: The more pixels that an image contains, the more level of detail it holds. Increasing the number of pixels (resolution) in an image increase rendering time [9]. We selected three common image resolutions:

- 1280 x 720.
- 1440 x 900.
- 1920 x 1080.

Effects: We used the 2^k [13] form for evaluating the impact of the effects: anti-aliasing, reflections, and transparencies. This significantly reduced the number of experiments while allowing us to assess if influences in both algorithms' performance. Visual effects directly increase rendering time [17]. All possible combinations of the following three effects were considered:

- AA: Anti-aliasing.
- RE: Reflection (5 levels).
- TR: Transparency (5 levels).

Acceleration algorithm: This is the most crucial factor of our experiment; the levels are:

- Kd-Trees.
- BVH.

There are 5 x 3 x 8 x 2= 240 combinations, as we decided to do 15 replications to increase the validity of the results, we ended up with 240 x 15=3600 runs of the experiment. We automated the execution and recollection of data trough scripts.

The equipment used to run the experiments is described in table 1.

Table 1. Hardware description.

Characteristic	Specification
Vendor	AMD
CPU Model	Ryzen 2600
Price($)	199
CPU Cores/Threads	6/12
Power Consumption(W)	65
CPU Cache L2/L3(MB)	3/16
CPU Frequency(GHz)	3.4-3.9
RAM (GB)	8
RAM Frequency (MHz)	2400
RAM Configuration	Single Channel
Storage Type	SSD
Storage Size (GB)	256

The scenes used for the experiments contained different objects randomly distributed through the *x, y, z* axis. We limited the random range to guarantee that all the objects were held inside the projection frame. We also randomized the sizes, shapes, and colors of the objects contained in the scene. In this way, we created a good representation of a ray-traced scene [15]. The factors of resolution, effects, and acceleration algorithm were adjusted to fit a specific combination of the experiment.

The response variable of the experiments is rendering time, as it represents performance [14].

Our research differs from [17], as they used an Accelerated Processing Unit (APU) to increase the performance of computationally expensive workloads. It also differs from [16] as they make a performance evaluation of the classic and BVH accelerated ray tracing algorithm using all the computing resources available in commodity hardware solutions.

Results

To comply with ANOVA adequacy requirements, we applied a square root transformation of the response variable [13]. Nonetheless, we present results de-transformed.

The results of the ANOVA analysis produced by R [20] for this experiment are shown in table 2.

Table 2. Anova table.

Characteristic	Sum Sq	Df	F value	Pr (>F)
Algorithm	99706	1	41426901.1	2.2e-16
Objects	376757	4	39134663.2	2.2e-16
Resolution	78398	2	12286865.3	2.2e-16
Effects	1211889	7	71932539.9	2.2e-16
Algorithm:Objects	5744	4	596614.8	2.2e-16
Algorithm:Resolution	6048	2	1256371.1	2.2e-16
Algorithm:Effects	226974	7	3993662.3	2.2e-16

Average rendering time in function of the acceleration algorithm is presented in figures 6, 7, 8 and 9 presents the average rendering time per effects, objects and resolution factors in function of the acceleration algorithm.

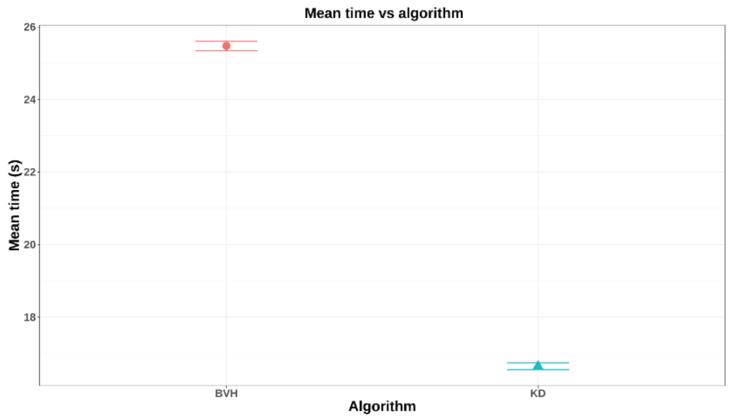

Figure 6. Average rendering time in function of the acceleration algorithm.

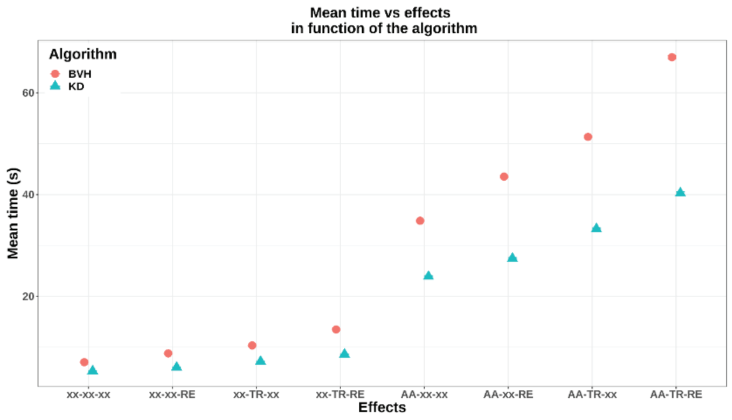

Figure 7. Average rendering time per effects in function of the acceleration algorithm.

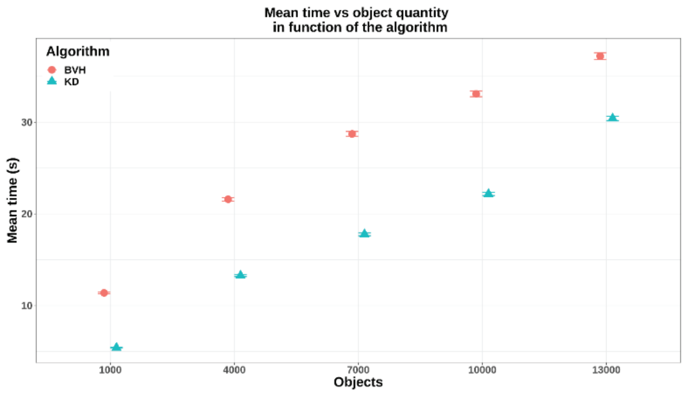

Figure 8. Average rendering time per object quantity in function of the acceleration algorithm.

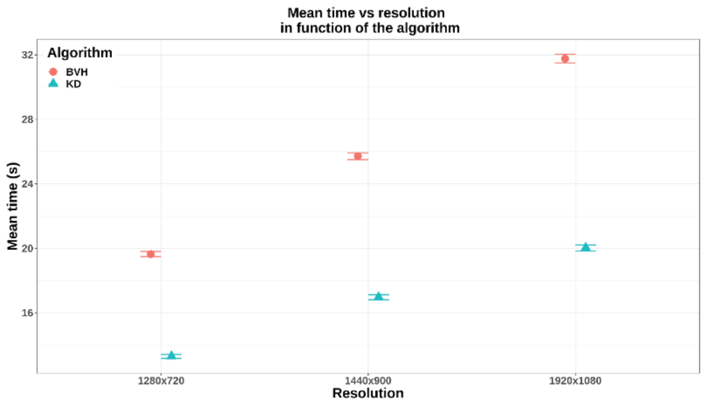

Figure 9. Average rendering time per resolution in function of the acceleration algorithm.

The average rendering time of each acceleration algorithm is displayed in table 3. This time includes and summarizes all combinations of effects, resolution, and object quantities.

Table 3. Obtained metrics.

Characteristic	Specification
Algorithm	Average Time (s)
BVH	25.47
Kd-Trees	16.65

Two sample images generating through the experiments are found in figures 10 and 11. Both images were rendered at the maximum resolution with all the effects.

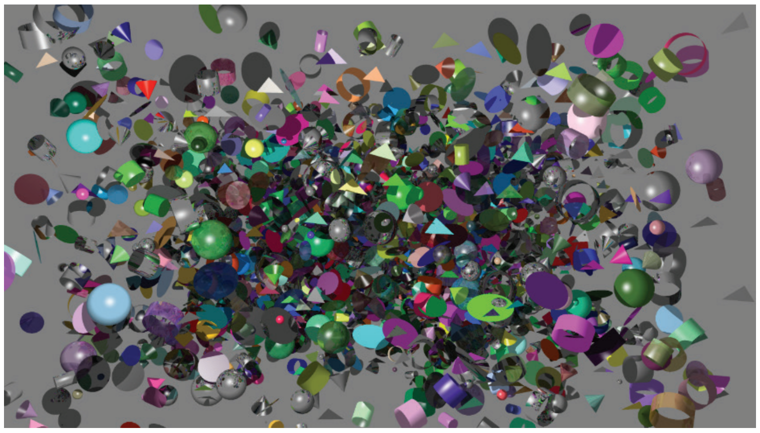

Figure 10. Scene with 7000 randomly distributed objects.

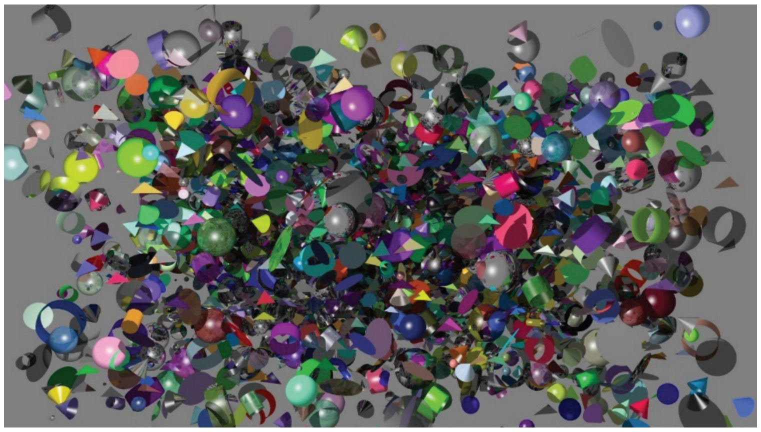

Figure 11. Scene with 10000 randomly distributed objects.

Discussion

Table 2 shows that the factors defined for this research and its interactions are statistically different (as the low obtained *p* values demonstrates). With this information, we can conclude from the average rendering times that we obtained during the experiment.

The average rendering time per algorithm is summarized in Figure 6. The figure shows that for the defined experiment, Kd-Trees performs significantly better than BVH.

We proceeded to decompose the rendering time per each one of the factors defined for the experiment as it is shown in figures 7, 8 and 9. In all cases we found the same result, Kd-Trees has better performance than BVH.

For the case of effects in function of the acceleration algorithm, in figure 7, we see a tendency that as more effects are included in the image, the rendering time difference between BVH and Kd-Trees becomes larger. When an effect like anti-aliasing is included in a scene, it generates more rays and thus more mathematical operations and memory accesses during rendering [1]. More rays and memory accesses mean more traversals of the BVH and Kd-Trees binary trees. As we see from the results, Kd-Trees perform better when these two factors are increased.

For the case of object quantity in function of the acceleration algorithm, we found out that in all instances, Kd-Trees perform better. Still, we don't see any tendency as object quantity increases. Object quantity is directly related to the size of the BVH and Kd-Trees trees, and thus is directly related to memory accesses [15]. As our results showed, it seems that KD-Trees performs better than BVH when memory accesses are increased.

Increasing the resolution of an image, adds more mathematical operations and memory accesses, as is displayed in figure 9. Again, Kd-Trees performed better than BVH for all resolutions. We observed that as the number of pixels increased, the advantage of the Kd-Trees accelerator over BVH became larger.

Modern x86 CPUs are more efficient as they have a better internal design. For instance, modern CPUs have fewer pipeline stalls, more cache memory, better memory access mechanisms, and improved branch prediction than older processors [8]. All these improvements could favor Kd-Trees in modern architecture. As we saw, for all defined scenarios, Kd-Trees performed better than BVH. Nonetheless, this is not the preferred acceleration technique in modern renderers [1,15].

Trade-offs between BVH and Kd-Trees are well documented [15]; for instance, BVH is more efficient building the tree than Kd-Trees, and Kd-Trees deliver slightly faster ray intersection tests. From our experiment, we can observe in table 3 the average rendering time of both algorithms, in which kD-Trees is 34% faster than BVH. In this case, Kd-Trees is considerably faster, not slightly.

Conclusions and Future Work

We explored the performance of the BVH and Kd-Trees algorithm in a modern x86 CPU. Our tests showed favorable results for the Kd-Trees acceleration structure in all tests. Our ray tracer included anti-aliasing, reflections, transparencies, different primitives and resolutions.

This research provided an up-to-date comparison of both algorithms in a modern processor. We implemented the ray tracer using the C programming language and did our implementations of both algorithms as fair as possible. Legacy x86 architectures are quite different from modern x86 processors as they have more cache memory, better memory access systems, reduced pipeline stalls, and improve prediction mechanisms [8]. These factors could favor the KD-Trees ray/object intersection mechanism.

For future work we would like to explore more deeply the reason of the advantage of Kd-Trees over BVH, in this case we would like to explore memory access patterns, processor affinity and cache misses in both modern and legacy (at least 15 years older) x86 processors. Also, we would like to perform the tests in different programming languages, as different previous evaluations were performed in GPUs with CUDA [19], others were performed in C++ with legacy processors [7], so we would like to discard programming language as a driving factor for the performance differences. Finally, we would like to verify if the advantage of KD-Trees vs BVH remains in different CPU architectures like ARM, RISC-V and MIPS.

References

[1] T. Akenine-Möller, E. Haines, and N. Hoffman, Real-Time Rendering, Fourth Edition. A K Peters/CRC Press, 2018.

[2] J. Buck, The Ray Tracer Challenge: A Test-Driven Guide to Your First 3D Renderer (Pragmatic Bookshelf). Pragmatic Bookshelf, 2019.

[3] Chitalu, Floyd M. and Dubach, Christophe and Komura, Taku, “Bulk-synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs,” in Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D ’18. New York, NY, USA: ACM, 2018, pp. 4:1–4:9.

[4] B. Choi, B. Chang, and I. Ihm, “Improving memory space efficiency of kd-tree for real-time ray tracing,” Computer Graphics Forum, vol. 32, 10 2013.

[5] P. Du, E. S. Liu, and T. Suzumura, “Parallel Continuous Collision Detection for High-performance GPU Cluster,” in Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ser. I3D ’17. New York, NY, USA: ACM, 2017, pp. 4:1–4:7.

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

[7] V. Havran, “Heuristic Ray Shooting Algorithms,” Ph.D. dissertation, Czech Technical University, 166 36 Prague 6, Czechia, 2000.

[8] J. Hennessy, Computer Architecture: A Quantitative Approach. Cambridge, MA: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018.

[9] J. F. Hughes, A. van Dam, M. McGuire, D. F. Sklar, J. D. Foley, S. K. Feiner, and K. Akeley, Computer Graphics: Principles and Practice (3rd Edition), 3rd ed. Pearson India, 2019.

[10] M. R. Kaplan, “The use of spatial coherence in ray tracing,” 1987.

[11] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," in Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 269–278.

[12] J. P. Molina Masso and P. González, "Automatic hybrid hierarchy creation: a cost-model based approach," Computer Graphics Forum, vol. 22, pp. 5–13, 03 2003.

[13] D. C. Montgomery, Design and Analysis of Experiments. Tenth Edition. Wiley, 2020.

[14] D. Patterson, Computer Organization and Design: The Hardware/Software Interface. Sixth Edition. Morgan Kaufmann, 2020.

[15] M. Pharr, W. Jakob, and G. Humphreys, Physically Based Rendering: From Theory to Implementation, 4th ed. Early Release, Morgan Kaufmann, 11 2022.

[16] E. Rivera-Alvarado and F. J. Torres-Rojas, "Ray tracing acceleration through heterogeneous integrated commodity hardware," in 2019 38th International Conference of the Chilean Computer Science Society (SCCC), 2019, pp. 1–8.

[17] E. Rivera-Alvarado and F. J. Torres-Rojas, "Apu performance evaluation for accelerating computationally expensive workloads," Electronic Notes in Theoretical Computer Science, vol. 349, pp. 103 – 118, 2020, proceedings of CLEI 19, the XLV Latin American Computing Conference.

[18] E. Rivera-Alvarado and F. J. Torres-Rojas, "Bounding volume hierarchy acceleration through tightly coupled heterogeneous computing," in High Performance Computing, J.L. Crespo-Mariño and E. Meneses-Rojas, Eds. Cham: Springer International Publishing, 2020, pp. 94–108.

[19] 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, ser. SCCG '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 29–36.

[20] H. Wickham and G. Grolemund, R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media, 2017.

[21] H. Ylitie, T. Karras, and S. Laine, "Efficient Incoherent Ray Traversal on GPUs Through Compressed Wide BVHs," in Proceedings of High Performance Graphics ser. HPG '17. New York, NY, USA: ACM, 2017, pp. 4:1–4:13.

[22] Seacord, Robert, Effective Cs: an introduction to professional C programming, No Starch Press, Inc, 2020.