Data Structures in Ray Tracing Algorithm: A Literature Review

Mingqi (Tom) Geng
Princeton Int'l School of Math and Science
Princeton, United States
tom.geng@prismsus.org

Junda Wang
Peking University
Beijing, China
2200013111@stu.pku.edu.cn

Haochen Zheng
Shenzhen School of International Education
Shenzhen, China
orickzheng@outlook.com

***Abstract*—Ray tracing is an advanced technique utilized in modern video games to enhance players' immersive experience by creating realistic and lifelike scenes. Ray tracing requires a massive amount of calculation, thus, various algorithms, data structures, and hardware have been developed to optimize ray tracing. This literature review aims to provide a basic understanding of the fundamental data structures used in ray tracing, such as bounding volume hierarchies (BVH), octrees, and kd-trees. The review focuses primarily on the construction, traversal, and intersection testing capabilities of these structures.**

***Keywords*—Ray tracing, Data structures**

I. Introduction

Ever since the first video game, Tennis for Two, was made, game developers have tried their best to attract players by offering a unique and exciting gameplay experience for their target audience. While gameplay has always been a key factor, game developers have also recognized the importance of graphics in creating a more immersive experience for players. As a result, video games have evolved from simple 2D pixelated graphics to more realistic 3D graphics, which allows players to immerse themselves in the gaming world and experience it from the perspective of the characters they are playing. The use of the Tyndall effect in Red Dead Redemption 2, a video game released in 2010, is an excellent demonstration of the progress made in computer graphics to create a more realistic environment.

Ray tracing algorithm, originally developed in the 1970s [1], is widely used in modern video games to create photorealistic scenes. To optimize this algorithm, developers have focused on improving the speed of accessing 3D objects, eliminating unnecessary calculations of intersections, and reusing rendered pixels from previous frames. This is where efficient data structures come into play. Without an efficient data structure, ray tracing may have a negative impact on gameplay, as it cannot achieve real-time global illumination.

This review covers several data structures, including kd-trees, bounding volume hierarchies (BVH), and octrees, and examines their advantages and disadvantages. By doing so, we hope to provide insight into their performance trade-offs and potential applications in game design.

II. Background

Global illumination is a rendering technique that captures both direct and indirect light reflected and refracted in the environment. This results in a more realistic representation of lighting in a scene. Ray tracing is a commonly used method to achieve global illumination.

Ray tracing is a technique used in computer graphics to simulate the behavior of light. It is based on the principles of geometrical optics, which models light as rays that travel in straight lines and carry simple physical properties such as intensity and color. However, other properties, such as the phase of light, are ignored in this model.

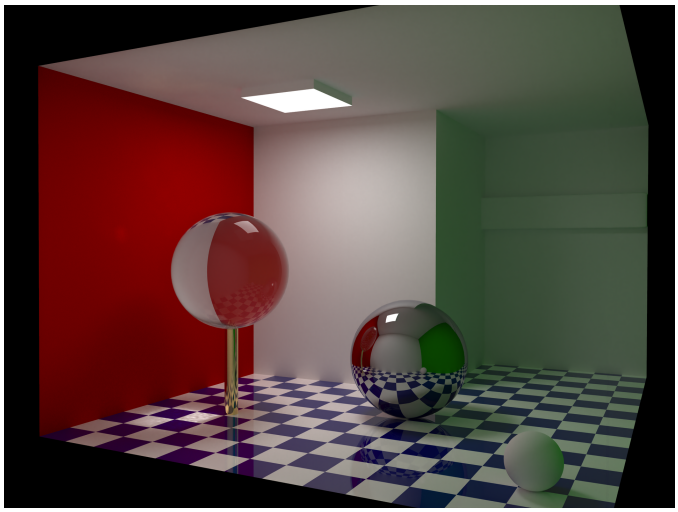

Fig. 1. Example of global illumination. Image from wikipedia.

The process starts by emitting one or more rays from the camera (eye position) through each pixel in an image plane. The nearest intersection between the ray and all the objects in the 3D scene is then calculated, as illustrated in figure 2. Depending on the properties of the material, the color and intensity of the light at that point is calculated and one or more reflection rays are sampled. This process is repeated until a maximum depth is reached. Finally, the intensity of the ray is traced back along its path according to the rendering equation [6].

$$L_o(\vec{p},\vec{\omega}_o) = L_e(\vec{p},\vec{\omega}_o) + \int_\Omega f(\vec{p},\vec{\omega}_i,\vec{\omega}_o)L_i(\vec{p},\vec{\omega}_i)(\vec{\omega}_i \cdot \hat{n})\mathrm{d}\omega_i \quad (1)$$

Where L_o is the radiance of the ray directing outward point $\vec{p}$ along direction $\vec{\omega}_o$, L_e is the radiance emitted by the material, L_i is the incoming ray from the environment to point $\vec{p}$, and f is the reflection function that only depends on the material of the reflecting surface.

III. Data Structures

A. Octree

1) Octree Structure and Construction: Octree is a hierarchical data structure commonly used in computer graphics and 3D computational geometry for spatial partitioning as shown in figure 3. A node in the octree represents a cubic volume of space and has eight child nodes, each representing one-eighth of the parent node's volume. Octrees can be used to efficiently store and query spatial information, making them well-suited for ray tracing applications.

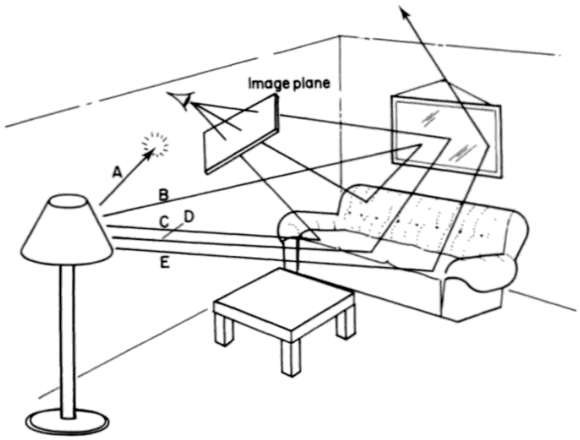

Fig. 2. A demonstration of ray tracing.
Image from *An Introduction to Ray Tracing* [4]

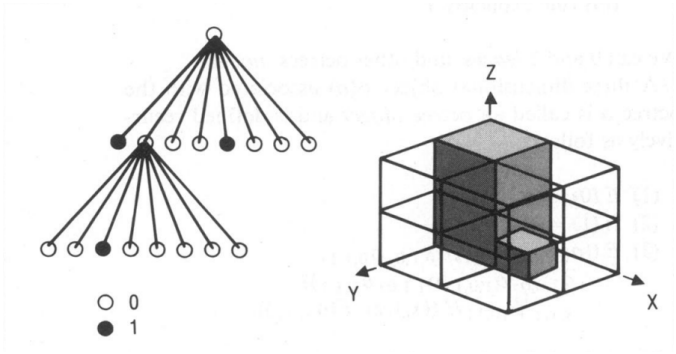

Fig. 3. A demonstration of the octree.
Image from *Octree-Related Data Structures and Algorithms* [3]

The construction of an octree begins by defining a root node that encompasses the entire scene. The space within the root node is then recursively subdivided into eight smaller cubic volumes until a termination condition is met, such as reaching a maximum depth or having a small number of primitives in a leaf node. The geometry within each leaf node is stored as a list of primitives or references to the original geometry. By ensuring a maximum number of objects in the leaf node, the octree can divide into finer grids in dense areas and larger grids in sparse areas.

In certain variations of the octree, non-leaf nodes can also store objects such that each object is the child of the smallest node that completely encloses its volume, rather than all nodes that intersect with the object. This ensures that each object has exactly one parent node.

2) Ray Traversal and Intersection Testing: In ray tracing, octrees are used to accelerate intersection tests between rays and scene geometry by only testing leaf nodes that potentially intersect with the ray, which are known as ray cells. There are multiple methods to traverse all ray cells in the octree [2].

- One of the most common traversal processes begins by testing the ray against the bounding volume of the octree's root node. If no intersection occurs, further traversal is unnecessary as the ray does not intersect any geometry in the scene. If an intersection is found, the ray is then tested against the bounding volumes of the child nodes, recursively descending the tree until reaching the leaf nodes containing the actual geometry. When a leaf node is reached, the ray is tested for intersection against the geometry primitives stored in that node. Relevant information such as intersection points, surface normal, and material properties can be recorded for further processing. The traversal process continues until all relevant nodes have been visited or a termination condition is met, such as finding the closest intersection point or reaching a maximum allowed traversal depth.
- Another method starts from the first leaf node the ray's origin lies in. Each time, all the objects stored in the node are tested for intersection with the ray. If an intersection is found, the algorithm stops and the closest intersection is returned. Otherwise, the algorithm tests the intersection between the ray and the boundary of the current cell and moves to the next cell adjacent to the current cell along the ray's path. The algorithm terminates when an intersection is found or no adjacent leaf nodes exist. This algorithm, known as the ray generator [5] [2], can be more time efficient than the previous method since the algorithm may terminate early after finding an intersection, and finding neighbors of cells generally has a lower time cost than traversing the tree [7]. Figure 4 demonstrates the ray generator algorithm.

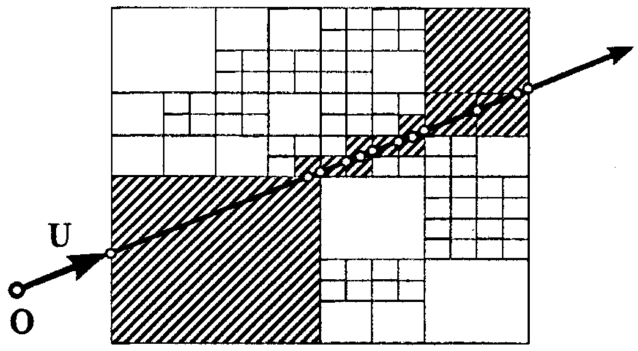

Fig. 4. ray generator for quadtree (2D version of octree).
Image from *Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing* [2].

The efficiency of octrees in ray tracing stems from their ability to quickly discard large portions of geometry that are not intersected by a ray. By traversing the tree and testing for intersections against bounding volumes, the number of intersection tests against actual geometry is significantly reduced compared to testing against all geometry in the scene.

3) Suitable Scenarios Analysis: By dividing the 3D space into smaller cubic cells, octrees can effectively reduce the number of objects that need to be tested for intersection, resulting in significant efficiency improvement. Moreover, octrees do not require prior knowledge of the distribution of objects in the scene, which allows them to be constructed quickly and efficiently.

However, octrees can have a higher memory overhead as they require storing eight child pointers per internal node, as well as additional node information such as bounding volumes or pointers to parent nodes. Besides, in scenes with uneven geometry distribution or high-depth complexity, octree traversal can be less efficient as uniform partitioning may lead to deep trees with many empty nodes.

Based on these strengths and weaknesses, octrees are particularly well-suited for scenarios where the scene has a relatively uniform distribution of geometry and a high degree of spatial coherence. However, they may not be the best choice for scenes with highly irregular or complex geometry, as the structure of the octree may become too complex to be useful.

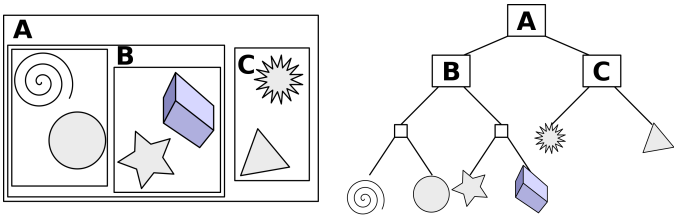

Fig. 5. A demonstration of BVH. Image from wikipedia.

B. KD-Tree

1) KD-Tree Structure and Construction: Kd-trees are a widely used data structure in computational geometry and computer graphics, particularly for nearest-neighbor searches and ray tracing. A kd-tree is a binary tree that recursively partitions space into two subspaces along one of the coordinate axes. Each node in the tree represents a splitting plane and its two children represent the resulting subspaces.

To construct a kd-tree, the first step is to choose a splitting plane that divides the scene into two parts with a roughly equal number of basic objects in each part. This process is repeated recursively for each child node until a termination condition is met, such as reaching a maximum depth or having a small number of primitives in a leaf node.

2) Advantages and disadvantages: As kd-trees' splitting planes can be positioned anywhere along the axes, they adapt well to the spatial distribution of geometry in the scene. This adaptability leads to efficient partitioning, which can improve traversal performance. Moreover, kd-trees' application is not limited in 3D and ray tracing, making them a versatile data structure in computer graphics and computational geometry.

Nevertheless, updating the kd-tree for dynamic scenes can be challenging, as moving or modifying geometry may require tree restructuring and reinsertion of objects. This can be computationally expensive and may limit its applicability in real-time applications.

C. Bounding Volume Hierarchy

1) BVH Structure and Construction: A BVH is a tree data structure that organizes geometric objects in a hierarchical manner, using bounding volumes to enclose groups of objects [9]. The BVH consists of two types of nodes: internal nodes and leaf nodes. Each internal node represents a bounding volume containing its child nodes, while leaf nodes store references to the actual geometric objects in the scene.

The choice of bounding volume is crucial for the performance of BVH, as it affects traversal efficiency and memory overhead. Commonly used bounding volumes include axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs). AABBs are generally preferred due to their simplicity and lower computational cost during intersection tests. In rare cases, bounding spheres are used due to their simplicity of computation and less difficulty in code.

The construction of a BVH involves partitioning the scene geometry into smaller subsets recursively, creating a hierarchical structure. The first step is choosing a suitable partitioning axis and splitting criterion, dividing the objects into two groups based on the chosen axis and criterion, and then creating a bounding volume for each group, encompassing all objects within the group. This process is repeated recursively for each child node until a termination condition is met, such as reaching a maximum depth or a minimum number of objects per leaf node.

There are several heuristics for selecting the partitioning axis and splitting criteria, such as spatial median, object median, and surface area heuristics, as shown in figure 6.

- Spatial median heuristic: Partition so that each child has equal volume.

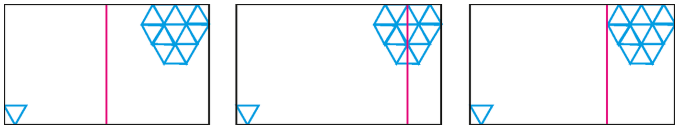

Fig. 6. Methods of partition. From left to right are the space median heuristic, object median heuristic, and surface area heuristic.
Image from median.com

- Object median heuristic: Partition so that each child contains an equal number of objects.

The next section will explain surface area heuristics (SAH) in detail.

2) Surface Area Heuristics (SAH): The Surface Area Heuristics (SAH) partitions the bounding volume based on the total surface area of all objects and the number of objects in the volume. A set of objects $A = \{a_1, a_2, \cdots, a_n\}$, with the bounding volume of surface area S, is partitioned into two sets A_1 and A_2, whose bounding boxes' surface areas are S_1 and S_2, then the cost function is written as equation 2 [8].

$$C = \frac{S_1}{S} t_{\mathrm{t}1} + \frac{S_2}{S} t_{\mathrm{t}2} + \frac{S_1}{S} \sum_{a \in A_1} t_{\mathrm{i}\,a} + \frac{S_2}{S} \sum_{a \in A_2} t_{\mathrm{i}\,a} \tag{2}$$

Where $t_{\mathrm{t}\,n}$ is the estimate of the time it takes to traverse the nth sub tree (where $n = 1, 2$), $t_{\mathrm{i}\,a}$ is the estimated time to compute the intersection between a ray and the object a. Equation 2 can also generalize to the partitioning of multiple subsets.

If we assume that the rays have an equal probability of coming from any direction, the conditional probability of the ray intersecting with bounding volume S_1 given the ray intersects with S is the ratio of S_1 and S's average projection area. If we further assume that the bounding volumes are convex, then the average projection area is equal to $1/4$ of the total surface area [10]. That is:

$$P(S_1|S) = \frac{S_1/4}{S/4} = \frac{S_1}{S} \tag{3}$$

A similar equation can be written for S_2.

$$P(S_2|S) = \frac{S_2}{S} \tag{4}$$

Thus, the cost function of SAH on node A represents the expectation of time cost, including the cost to traverse the sub-trees and to check for intersections, given that the ray intersects with its bounding volume, S. SAH is designed such that after partitioning, the area with the most number of objects will get the least amount of volume, ensuring a minimum expectation time to test the intersection.

3) Advantage and disadvantage: BVH structures, similar to kd-trees, can be computationally expensive to build, particularly for dynamic scenes where frequent hierarchy updates are required. However, by using heuristics like the Surface Area Heuristic (SAH), the construction of BVH can be relatively faster.

Additionally, BVH structures can be traversed in parallel, making them ideal for GPU-based ray tracing implementations that leverage the massively parallel processing capabilities of modern GPUs. We will discuss parallel construction in the following section.

IV. Optimizations

A. Parallel Construction

Parallel construction techniques aim to leverage the computational power of modern hardware such as multi-core CPUs and GPUs, to accelerate the construction process, and enable real-time applications in gaming and other interactive scenarios.

The construction of octree, BVH, and other data structures in ray tracing also benefit from the development of hardware. Octrees are

a hierarchical data structure that can be constructed using a bottom-up approach, where objects are inserted into the tree and the tree is partitioned recursively. To parallelize this process, one can divide the scene geometry into smaller subsets and assign each subset to a separate processing unit (e.g., a CPU core or GPU thread). Each processing unit constructs a local octree for its assigned subset, and these local octrees are then merged into a global octree.

Similar to octrees, BVH construction can also be accelerated by parallel techniques. One popular approach for parallel BVH construction is the use of binning, where the scene geometry is divided into spatial bins along the partitioning axis. Each bin is processed independently by a separate processing unit, which calculates the optimal split position and bounding volume for its assigned bin. Once all bins have been processed, the results are combined to form the final BVH structure.

This approach takes advantage of the inherent parallelism in the construction process and significantly reduces construction time. It is essential for real-time applications in interactive and gaming scenarios, where rapid updates to data structures are required to accommodate dynamic scenes and user interactions.

B. Octree-R

Octree-R is an octree-variant data structure introduced by Kyu-Young Whang in 1995[11]. The prior objective of octree-r is to reduce the number of ray-object intersection tests. This is achieved by dividing the space using the plane that results in the least estimated number of ray-object intersection tests to create an octree, rather than using a spatial median to divide the space.

We should first note the cost model[11] for ray tracing using octrees, which is presented in the equation5 as follows:

$$Cost = n_v \cdot T_v + n_t \cdot T_i \tag{5}$$

Where n_v represents the number of voxels the ray has passed, T_v represents the time for the ray to move to the next voxel, n_t represents the number of ray-object intersection tests, and T_i represents the time taken for a ray-object intersection test. The equation indicates that if we assume the same number of voxels in the octrees to be compared, the one having the smallest value for n_t will have the smallest time cost.

Suppose that volume A contains volume B. Similar to SAH, we use the same estimate of $P(B|A)$ as shown in equation 3, which stands for the conditional probability for a ray to pass volume B when the ray passes volume A. Based on this equation, we can now estimate the expected number of ray-object intersection tests by applying it when we divide a given space R as shown in figure 7.

The conditional probability for a ray to pass through the space A once it passes through the space R is as follows:

$$P(A|R) = \frac{S_A(t)}{S_R} = \frac{2(tb + tc + bc)}{2(ab + ac + bc)} = \frac{t(b + c) + bc}{a(b + c) + bc} \tag{6}$$

A similar equation can be written for space B:

$$P(A|R) = \frac{S_B(t)}{S_R} = \frac{(a - t)(b + c) + bc}{a(b + c) + bc} \tag{7}$$

Now let us denote the number of objects intersecting with plane t in figure 7 as $s(t)$, the number of objects completely included in the space A as $n(t)$, and the number of objects completely included in the space B as $m(t)$. Then, the expected number of ray-object intersection tests can be given as follows:

$$E(t) = \frac{S_A(t)}{S_R} n(t) + \frac{S_B(t)}{S_R} m(t) + \left\{ \frac{S_A(t)}{S_R} + \frac{S_B(t)}{S_R} \right\} s(t) \tag{8}$$

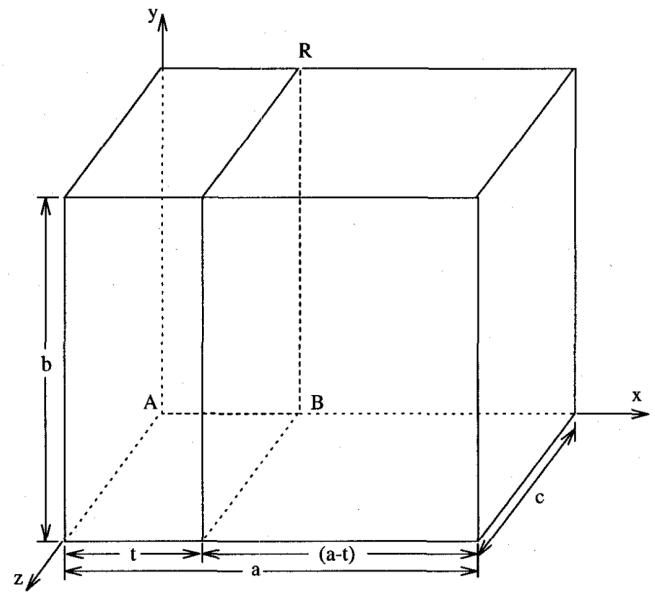

Fig. 7. Finding a proper dividing plane
Image from *Octree-R: Adaptive Octree for Efficient Ray Tracing* [11]

We can then substitute equation 8 with equation 6 and equation 7 and obtain:

$$\begin{aligned} E(t) = {} & \frac{t(b + c) + bc}{a(b + c) + bc} n(t) \\ & + \frac{(a - t)(b + c) + bc}{a(b + c) + bc} m(t) + \frac{a(b + c) + 2bc}{a(b + c) + bc} s(t) \end{aligned} \tag{9}$$

MacDonald and Booth [8] identified in 1990 that the optimal splitting plane lies between the spatial median and the object median. In order to divide the space into eight voxels, we should find the t planes for each axis of X, Y, and Z, which minimize the equation 9. This process can be accomplished by traversing several points equally spaced between the object median and the spatial median. The number of points should be decided through tests. By recursively repeating the process, we obtain an octree-R, which provides a performance gain of 4% to 47% over the conventional octree in the experiment[11].

V. Conclusion

In conclusion, data structures such as octree, kd-tree, and BVH play an important role in enabling real-time ray tracing in modern video games. Each data structure has its own advantages and limitations in terms of construction, traversal, and intersection testing performance. Optimizations like parallel construction and surface area heuristics can help improve the efficiency of these data structures.

Current game engines have started to showcase ray tracing technologies with impressive visual effects, such as Unreal Engine 5's Nanite virtualized micropolygon geometry system, which demonstrate the potential of upcoming ray tracing technologies. By leveraging massive amounts of geometric detail, future games may be able to create highly detailed and complex virtual environments that feel lifelike.

However, there is still space for further improvement. Future ray tracing techniques will likely focus on accelerating performance through specialized hardware optimizations, more efficient data structures, and algorithmic improvements. As computational power continues to increase with advances in CPU and GPU technologies,

Fig. 8. Scene rendered by Unreal Engine 5
Image from docs.unrealengine.com

we may see fully path-traced global illumination become feasible for real-time rendering in the near future.

ACKNOWLEDGMENT

We would like to thank William Nace, our professor, for his help and support for this paper.

REFERENCES

[1] James F. Blinn and Martin E. Newell. "Texture and Reflection in Computer Generated Images". In: *Commun. ACM* 19.10 (Oct. 1976), pp. 542–547. ISSN: 0001-0782. DOI: 10.1145/360349.360353. URL: https://doi.org/10.1145/360349.360353.

[2] Robert Endl and Manfred Sommer. "Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing". In: *Computer Graphics Forum* 13.1 (1994), pp. 3–19. DOI: https://doi.org/10.1111/1467-8659.1310003. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.1310003. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1310003.

[3] K. Fujimura et al. "Octree-Related Data Structures and Algorithms". In: *IEEE Computer Graphics and Applications* 4.01 (Jan. 1984), pp. 53–59. ISSN: 1558-1756. DOI: 10.1109/MCG.1984.275901.

[4] Andrew S Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[5] Andrew S. Glassner. "Space subdivision for fast ray tracing". In: *IEEE Computer Graphics and Applications* 4.10 (Oct. 1984), pp. 15–24. ISSN: 1558-1756. DOI: 10.1109/MCG.1984.6429331.

[6] James T. Kajiya. "The Rendering Equation". In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), pp. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: https://doi.org/10.1145/15886.15902.

[7] Aaron Knoll et al. "Interactive Isosurface Ray Tracing of Large Octree Volumes". In: *2006 IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, pp. 115–124. DOI: 10.1109/RT.2006.280222.

[8] J. David MacDonald and Kellogg S. Booth. "Heuristics for ray tracing using space subdivision". In: *The Visual Computer* 6.3 (May 1990), pp. 153–166. ISSN: 1432-2315. DOI: 10.1007/BF01911006. URL: https://doi.org/10.1007/BF01911006.

[9] Steven M. Rubin and Turner Whitted. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes". In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 110–116. ISSN: 0097-8930. DOI: 10.1145/965105.807479. URL: https://doi.org/10.1145/965105.807479.

[10] Zachary Slepian. *The Average Projected Area Theorem - Generalization to Higher Dimensions*. 2012. arXiv: 1109.0595 [math.DG].

[11] Kyu-Young Whang et al. "Octree-R: an adaptive octree for efficient ray tracing". In: *IEEE Transactions on Visualization and Computer Graphics* 1.4 (1995), pp. 343–349. DOI: 10.1109/2945.485621.