Computer Science Extended Essay

Optimization of BVH Construction in Ray Tracing Applications

Research Question:

“To what extent does increasing the number of children impact the Surface Area Heuristic ratio and Build Time in Bounding Volume Hierarchies in Ray Tracing Applications”

Word Count: 3489

Table of Contents

1. Introduction

Computer graphics has been an ever-evolving field in computer science over the past few decades. One of the leading causes has been rendering multidimensional models more accurately by simulating how light functions in the real world, thus achieving photorealistic images with three-dimensional models.

This approach is called Physically Based Rendering (Pharr et al., 2023, #2). Engineers, architects, and scientists can benefit from this by creating realistic models of specific parts or structures, saving time and money by experimenting virtually with graphics. One way to achieve photorealism in computer graphics is through Ray Tracing, a technique for simulating light in computer graphics that simulates how light behaves in the real world to generate photorealistic images.

Ray tracing is a prolonged process requiring many resources. A way to reduce the time to render using ray tracing is to use acceleration structures such as a Bounding Volume Hierarchy (BVH), significantly reducing the rendering time for ray tracing applications. A BVH can be considered a sorting tree with child and parent nodes, except in three dimensions.

This essay will focus on how changing the number of child nodes in a BVH affects the Surface Area Heuristic Ratio and the build time of BVH. Through the analysis, the aim is to focus on the implications that efficient BVHs can have in Computer Science applications.

2. Background Theory

2.1 Ray Tracing

Ray tracing is a technique for simulating light in computer graphics that simulates how light behaves in the real world to generate photorealistic images. A ray is a path light follows in a virtual environment. In the real world, light particles emanating from light sources bounce off of objects, and some of them are thereafter captured by our eyes or cameras, which help us see physical objects. However, in ray tracing, this process is reversed; Each pixel on the screen sends a virtual ray until it bounces off surfaces, repeating until it hits light-emitting sources. This way, resources are minimized because, theoretically, light sources scatter rays all around us, and when done in a virtual environment, the computer has to calculate the light we cannot see from the image plane, thereby increasing the computational power required. Multiple rays are cast from the camera through each pixel on the image plane and travel in a specific direction until it hits an object in the scene. A vector represents this trajectory, as the ray has both numerical value and direction. Interaction with different objects includes reflection and refraction. Ray tracing focuses on the path of light rays, as shown in **Figure 1**.

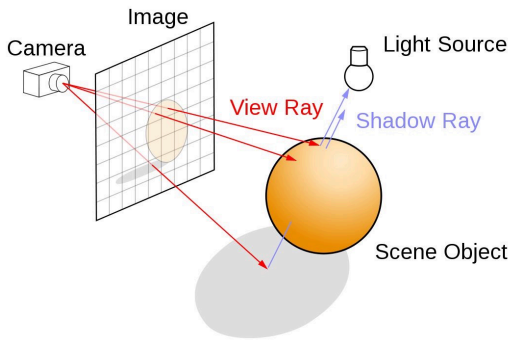


Figure 1: Visualization of ray tracing (Nvidia)

During ray generation, a ray is cast through each pixel on the image plane. The intersection of each ray with objects in the scene is then tested by checking all objects for intersections with the ray. This process is called Collision Detection. Once an intersection is detected, the coordinates are recorded. The efficiency of finding intersections between rays and objects greatly influences the rendering speed.

2.2 Collision Detection

Collision Detection is a crucial component of the ray tracing algorithm, as it determines all the intersections between rays and primitives in the scene (Dinas, 2009). The efficiency of collision detection significantly impacts the rendering time and results produced, thus making it essential for optimization in ray tracing applications. One way to efficiently achieve this is through Bounding Volume Hierarchies.

2.3 Bounding Volume Hierarchies

Bounding Volume Hierarchies or BVHs are tree structures that organize geometric objects where bounding volumes are recursively created. These objects are known as primitives. The term 'Bounding Volume' refers to encapsulated primitives bound in volumes such as boxes or spheres. This enables the computation of light interactions with these volumes, determining if they reach the leaf nodes (Physically Based Rendering, Pharr, 2023). This is demonstrated in two dimensions in **Figure 2**.

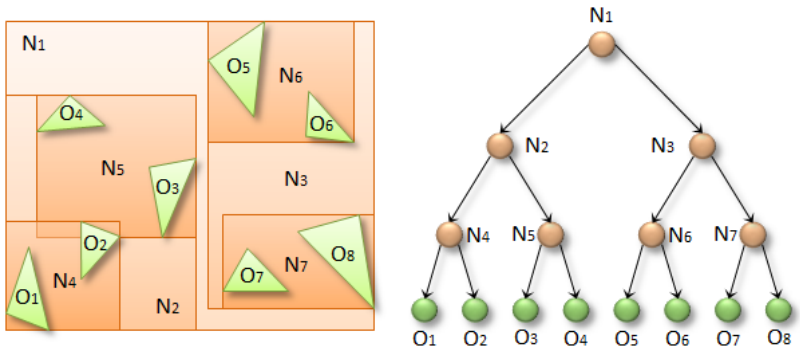


Figure 2: Visualization of BVH trees (Nvidia)

Here, *N* represents the bounding volumes, and *O* represents the primitives. Primitives in the context of ray tracing refer to basic geometric objects such as spheres, planes, triangles, and cylinders that interact with rays. Primitives can also be used to create more complex objects in a scene by combining or transforming them. As shown in **Figure 2**, primitives are usually triangles since three-dimensional objects can be constructed faster with triangles than any other shape (Küçükkarakurt, 2021). The boxes are there to encapsulate the primitives as efficiently as possible. The main idea is to intersect with parent nodes to reach child nodes. If we look at this tree from a binary search perspective, it can be seen that the *N* represents nodes, and *O* represents the leaves.

Now, consider a virtual ray approaching Bounding Volume $N1$ in Figure 2. Three possibilities can occur:

1) The ray can completely miss $N1$, thus bypassing all other volumes.
2) The ray can intersect with $N1$, but may not hit any primitives. It may continue to intersect with other volumes without hitting a primitive.
3) The ray can intersect with $N1$, intersect with another volume, and eventually hit a primitive.

Considering the second scenario, if a ray intersects with the outer bounding volume and also some of the inner bounding volumes without actually intersecting with a primitive, then some of the computing power is wasted since the algorithm has to note whenever a ray intersects with a bounding volume and if a ray does not intersect, the computer terminates the process. Unfortunately, creating these high-quality BVHs is difficult to parallelize and is computationally expensive, making them less appropriate for situations where the scene geometry is constantly changing. This restriction impacts the development of interactive apps. Due to this redundancy, it was logical to create algorithms that account for the probability of finding a primitive inside a box. Although many types of BVHs are used

in modern computing that may reach this potential to an extent, this essay will focus on the operational efficiency and optimization of an algorithm called Surface Area Heuristic BVH.

2.4 Construction of Bounding Volume Hierarchies

The BVH tree is constructed hierarchically, and inner nodes help organize and efficiently enclose groups of primitives, allowing for faster traversal during ray tracing. High-quality BVHs are typically built using the "greedy top-down sweep" technique, which is thought to be the most effective method for ray tracing. The "greedy top-down sweep" technique refers to optimizing the traversal of the ray-tree data structure by making a locally optimal choice at each intersection stage to find the most optimal solution (Md, 2023).

High quality in the context of Bounding Volumes refers to efficiency during tree construction. By enhancing previously developed, lower-quality BVHs after they are formed, some more recent techniques can produce comparable outcomes (Karras & Aila, n.d.). The goal is to minimize the cost and minimize memory usage. The cost of making a partition between two nodes can be notated as:

$$C = C_{trav} + p_A C_A + p_B C_B$$

Equation 1: Total cost of making a partition between nodes (Visual Computing Systems, 2009)

Where:

- C is the total cost
- C_{trav} is the cost of a ray traversing to an interior node
- C_A and C_B are the costs of intersection with the resultant child subtrees
- p_A and p_B are the probability that a ray will intersect with a bounding box of the child nodes A and B

2.5 Spatial Splits

Spatial splits are a technique used in the construction of BVH for interactive ray tracing. During the construction, the scene is divided into smaller bounding volumes until each Bounding Volume contains only a few primitives. One way to implement this is using Surface Area Heuristic splits.

2.6 Surface Area Heuristic BVH

Surface Area Heuristic (SAH) is an algorithm used in the construction of BVH trees for real-time/interactive ray tracing. It is a type of spatial split method that calculates the cost of splitting a node into Volumes A and B based on the ratio of the surface area of the bounding volumes and primitives. SAH is an algorithm used in the construction of BVH trees for real-time ray tracing. SAH is fundamentally a cost estimation function that calculates the cost of traversing a BVH tree based on the ratio of the surface area of the bounding volumes and primitives. The cost function predicts the cost of a defined split position on a per-node basis and is used to minimize the number of intersection tests, which can enable interactive ray tracing, where input and output are possible in real-time. The SAH is based on probabilities and is calculated as the sum of the cost of traversing the parent node, the cost of testing the shape against the ray, and the probabilities that the bounding volumes containing the children nodes intersect the ray. By minimizing the cost function, SAH can create a very efficient BVH tree structure which reduces the number of intersection tests required for real-time ray tracing (Physically Based Rendering, Pharr, 2023). One efficient way to achieve this is to utilize the SAH cost function. The SAH helps guide this construction by providing a quantitative measure to evaluate the efficiency of different split positions. The model of the equation states that:

$$C(A,B) = t_{traversal} + p_A \sum_{i=1}^{N_A} t_{intersect}(a_i) + p_B \sum_{i=1}^{N_B} t_{intersect}(b_i)$$

Equation 2: Equation for Surface Area Heuristic (Wiche, 2018)

Where

- $C(A, B)$ is the cost of splitting a node into volumes A and B
- $t_{traversal}$ is the time to traverse an interior node
- p_A and p_B are the probabilities that the ray passes through each volume
- N_A and N_B are the number of triangles in each volume
- a_i and b_i are the *i*th number of triangles in each volume (i.e. if $i = 4$, then a_i = 4th triangle)
- $t_{intersect}$ is the cost for one ray-triangle intersection.

The probabilities p_A and p_B can be computed by using this equation:

$$p(C|P) = \frac{S_C}{S_P}$$

Equation 3: the conditional probability that a ray passing through P will also intersect through C - retrieved from (Wiche, 2018)

Here:

- S_C and S_P are surface areas of volumes C and P, where the surface area of a node can be simply computed by summing all faces of a node) (Wiche, 2018)
- $p(C|P)$ is the conditional probability that a ray passing through P will also pass through C, where C is a convex volume in another convex volume P. (Wiche, 2018)

These two equations help convey that the lower the cost, the more efficient the computational process will be. The goal is to decrease the number of intersection tests needed during ray tracing, improving overall performance and rendering speed. **Figure 4** shows how utilizing the Surface Area Heuristic Ratio helps us lower the cost:

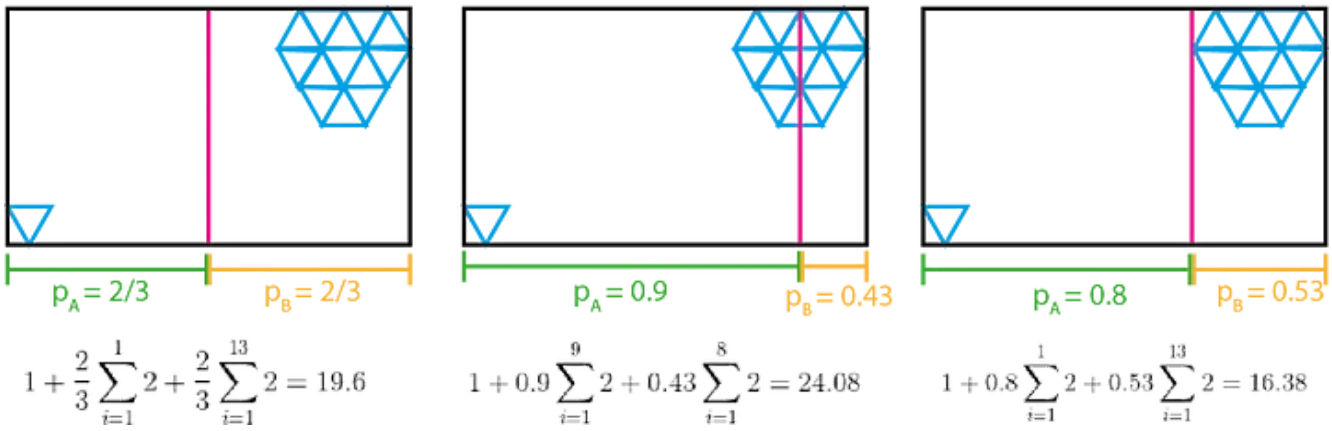


Figure 4: Practical application of the SAH equation (Wiche, 2018)

In Figure 4, Roman Wiche, the article's author, showed a practical example of this (Wiche, 2018). It can be seen that there are two objects constructed from triangles in this bounding volume. One is a single triangle on the bottom left, and one is thirteen triangles stacked together on the top right. We can say that these are small and large primitives respectively. Here there are three different scenarios where the topmost left diagram in Figure 4 with a line in the middle separates the two primitives. In the third case, we can see that the algorithm has divided the two primitives unequally. Although the probability of a ray entering subsection *A* is higher than *B*, the ratio of the surface area of subsection *B* to primitive is much larger than the ratio of the primitive in subsection *A* to the primitive in that section. This indicates that if the ray passes through subsection *B*, it has a higher chance of intersecting with the primitive situated in *B,* thus effectively decreasing the cost function. It can be verified that in **Figure 4**, the diagram on the rightmost has the lowest cost function $C(A, B) = 16.38$.

3. Experiment Methodology

Primary data was the main source of data for this experiment. An open-source kernel library called Intel Embree will be used to experiment with SAH and build times of the Bounding Volumes. Embree is a collection of high-performance ray tracing kernels that can be integrated into another software to achieve efficient real-time ray tracing. However, we can use the sample testing library provided within Embree. In one of the tutorial libraries in Embree, there is an executable tutorial file called bvh_builder, and the source code of it is written in the bvh_builder_device.cpp file, which contains a BVH algorithm. When run through the terminal, it gives the Build Time, SAH ratio, and how many primitives the ray hits in a second. See Appendix A1, which contains the process used for building Embree on the system to ensure authenticity. The maxBranchingFactor is a function in bvh_builder_device.cpp file which refers to the maximum number of children a BVH node can have. A .cpp file, for reference, contains a C++ source code, which contains all the main variables. This directly relates to the research question because by changing the maxBranchingFactor, we can get the desired output of ray intersection tests and SAH ratio. In ray tracing applications, there are very limited resources available in the amount of time given. So we tend to use only a normal quality constructed BVH, and only a limited number of rays are cast. Code Snippet 1 shows the argument of maxBranchingFactor.

```
/* settings for BVH build */
RTCBuildArguments arguments = rtcDefaultBuildArguments();
arguments.byteSize = sizeof(arguments);
arguments.buildFlags = RTC_BUILD_FLAG_DYNAMIC;
arguments.buildQuality = quality;
arguments.maxBranchingFactor = 2;
```

Code Snippet 1: argument.maxBranchingFactor

```
  RTCScene g_scene  = nullptr;
  ssize_t totalMemoryConsumed = 0;

  /* This function is called by the builder to signal progress and to
   * report memory consumption. */
    bool memoryMonitor(void* userPtr, ssize_t bytes, bool post) {
        if (!post)
            totalMemoryConsumed += bytes;
        return true;
}
```

Code Snippet 2: Memory function added into bvh_builder_device.cpp

```
std::cout << ", memory used: " << totalMemoryConsumed /
    1024 << " KB [DONE]" << std::endl;
```

Code Snippet 3:Memory Output added into bvh_builder_device.cpp

The BVH build process will be CPU-based, because of the accuracy it has. The computer used for testing is M1 MacBook Air with 8 gigabytes of RAM. The original code of the bvh_builder_device.cpp file does not contain a memory function, which was added by the author. This is shown in **Code Snippet 2** and **3**, and the entire modified code is shown in Appendix A3. We could not have used maxBranchingFactor = 1 because it would create a linear tree, which would be similar to a list, resulting in very inefficient collision detection. The worst case scenario of maxBranchingFactor = 1 would be $O(n)$ instead of $O(n(log(n))$. The result of the command ./embree_bvh_builder in the build folder is shown in Appendix A2 to provide a sample output of the code. The code provided in Appendix A3 constructs BVHs randomly in a limited area. This could have been prevented however it was not necessary because in interactive and real-time applications, the number of bounding volumes change according to lighting and what is visible in the image plane of the user.

4. Experimental Results

4.1 Data Sets Used

Table 1 shows the data retrieved from the bvh_builder. The 'maxBranchingFactor' is the maximum number of children a BVH node can have.

Low Quality BVH

maxBranchingFactor	Build Time (ms)	SAH	Memory (KB)
2	26.31987	265.844	159386
3	23.79359	36.3454	121883
4	22.7359	33.1975	107819
5	23.8506	13.7794	103132
6	23.44523	14.62233	98444
7	22.69803	4.72885	95318
8	22.47018	7.87197	90631

Medium Quality BVH

maxBranchingFactor	Build Time (ms)	SAH	Memory (KB)
2	98.5287	249.248	303148
3	97.03661	18.0228	229704
4	94.5498	15.1434	200014
5	94.88609	6.06528	190639
6	94.2925	14.1499	182825
7	96.0277	3.22335	175012
8	94.83719	9.09588	167199

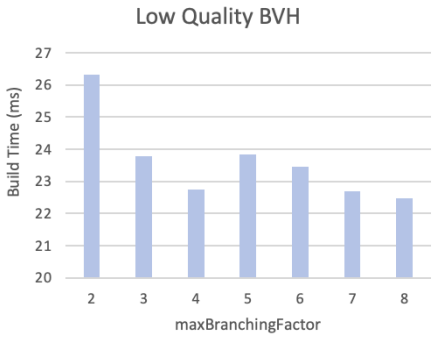
High Quality BVH

maxBranchingFactor	Build Time (ms)	SAH	Memory (KB)
2	172.8614	248.529	446909
3	168.7968	16.9986	337525
4	168.0226	15.057	292209
5	168.4489	6.07195	278146
6	165.6849	6.1522	267207
7	165.6771	3.21876	254706
8	165.2136	5.85927	243768

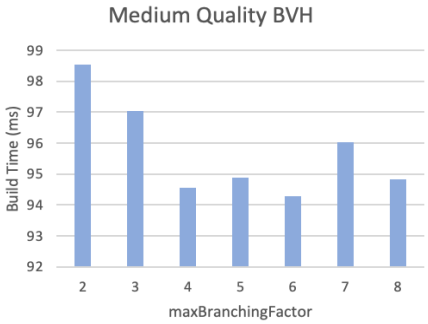
Table 1: Data for Low, Medium, and High-Quality Bounding Volume Hierarchies

4.2 Graphical Representation

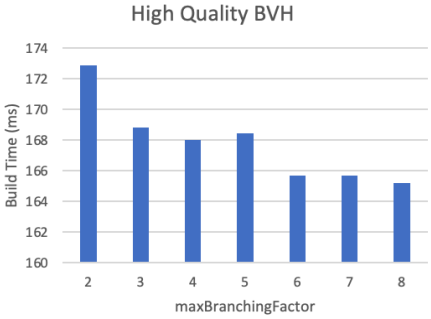
To better understand the trends in Surface Area Heuristics, Memory usage, and build times, a graphical representation will be used.



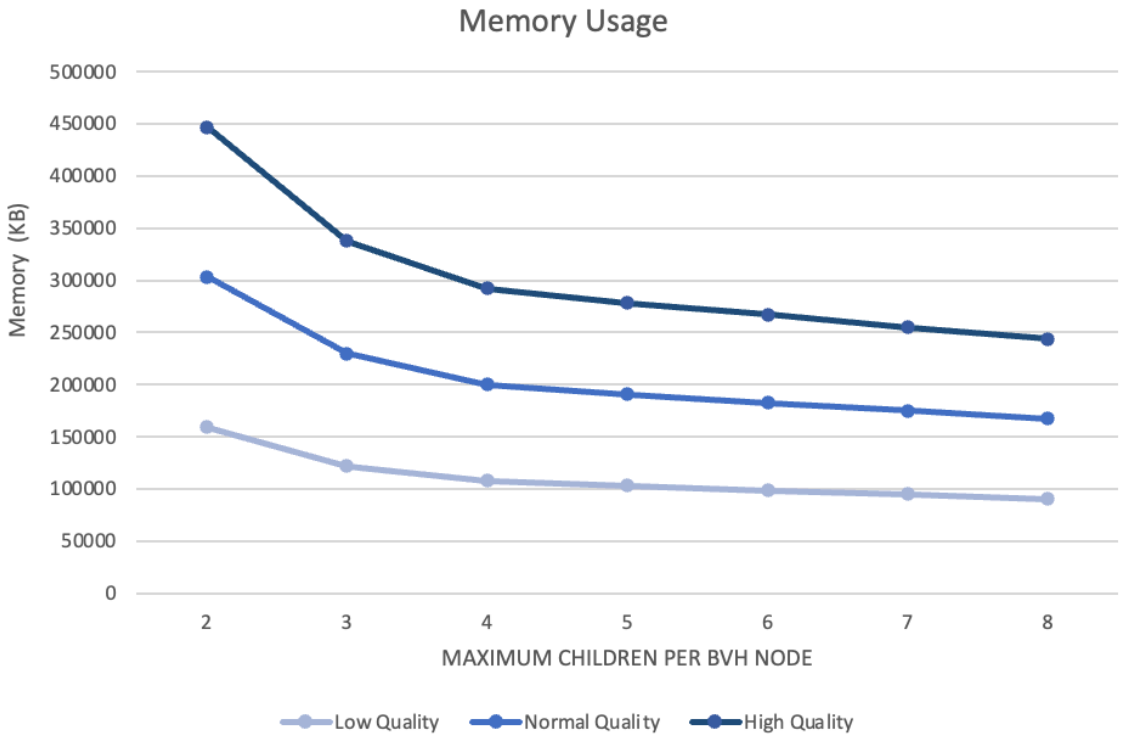
Graph 1: Low Quality BVH



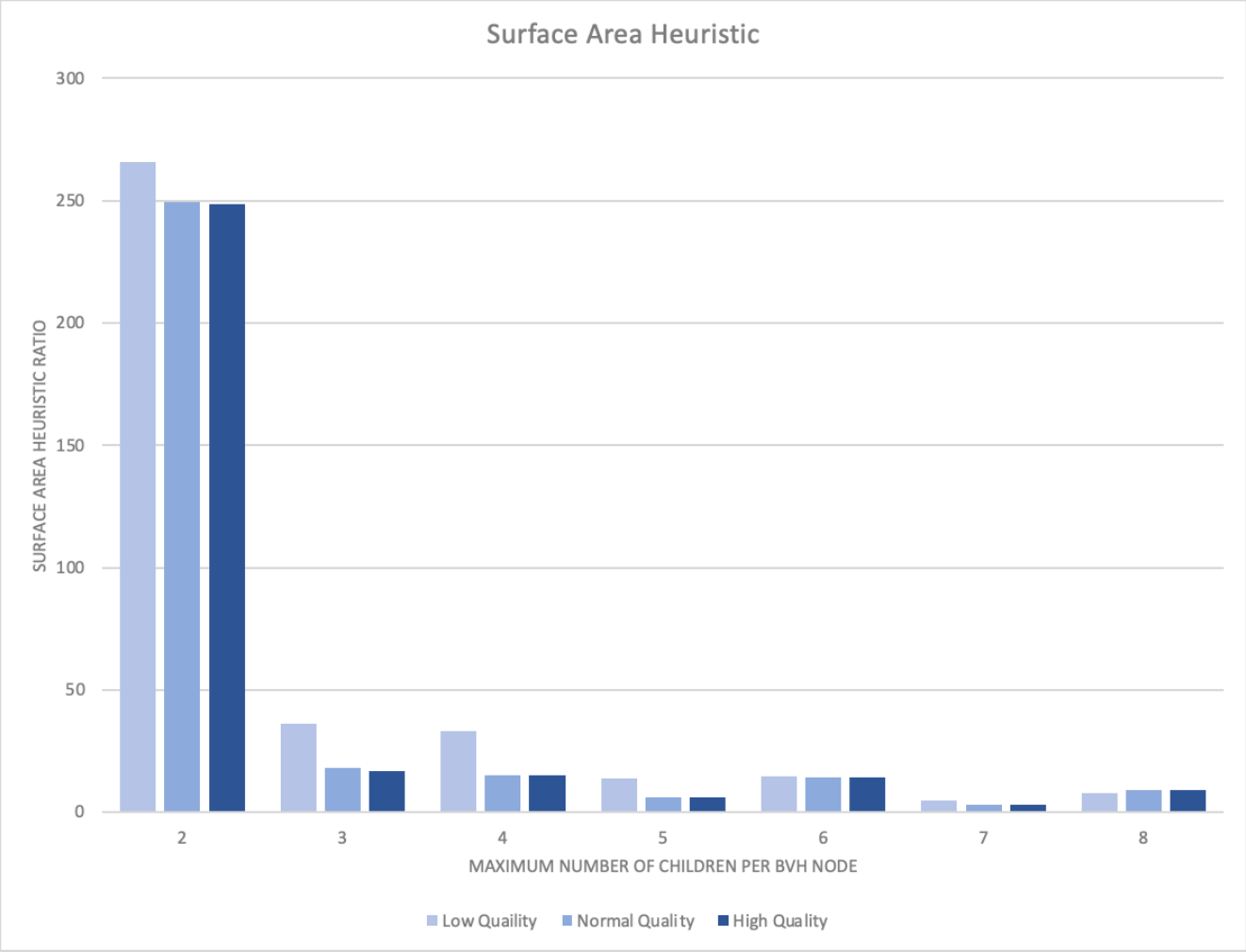
Graph 2: Med Quality BVH



Graph 3: High Quality BVH



Graph 4: Memory Usage of Low, Medium, and High-Quality Bounding Volume Hierarch



Graph 5: Surface Area Heuristic Ratio for all types of BVHs, lower is better

4.3 Result Analysis

Although the data for each is consistent, it has some discrepancies. It is to be noted the executable file of bvh_builder was run several times to make sure the results represent the real world scenarios, because the file bvh_builder_device.cpp generates bounding boxes randomly.

Memory Usage

Graph 4 shows the memory usage by all the BVH builds. The memory usage generally decreases as the maxBranchingFactor increases. This potentially suggests that the higher maxBranchingFactor values we use, the more memory-efficient BVH structure will be. This does not mean that we can increase the maxBranchingFactor indefinitely, because every different scene will have a different BVH, and we have to take into account that the ray intersection will be affected if the BVH has many children nodes. Memory usage also depends on the number of leaf nodes and the scene complexity, instead of just the children nodes.

Build Time

The graph of all types of BVH builds seems to be trending downwards. Utilizing the data above, it seems that Higher maxBranchingFactor value can lead to faster BVH construction times because larger nodes are created reducing the overall number of nodes in the BVH. In Graphs 1 and 2, for Low and Medium Quality BVHs, it can be seen that when maxBranchingFactor > 4 , The data seems to appear counterintuitive, since increasing the branching factor could be expected to decrease the times taken to build the BVHs. A logical explanation behind this could be that as the maxBranchingFactor increases, the depth of the tree increases as well, leading to traversal overhead during ray intersection tests, offsetting the benefits of reduced build times. Larger branching factors also result in larger nodes, which might not fit entirely within memory addresses, leading to counterintuitive results.

Surface Area Heuristic

SAH seemed to decrease as the maxBranchingFactor increased. The difference between when maxBranchingFactor was 2 compared to when it was 3 is quite a drastic difference. This is because when it was 2, the BVH construction algorithm had a more limited choice when splitting the nodes. This can result in inefficient splits, leading to higher SAH values. Overall, the SAH value dramatically decreased, which is a positive sign because the lower the SAH value the more likely the ray will intersect, reducing the computational requirements. Again, the difference between 2 and 3 can also be explained by memory constraints because the higher maxBranchingFactor, the more nodes the algorithm needs to create for the scene. This can lead to uneven node partitioning, leading to higher SAH values.

5. Potential Research Opportunities

Machine learning

Often in real time/interactive applications, machine learning is used to create the most efficient Bounding Volume Hierarchies possible. Machine Learning Algorithms can be trained to learn to optimally split heuristics based on factors such as hardware and scene complexity. Since BVH construction involves several parameters, ML algorithms can be trained to set those factors in real time to ensure the most efficient BVH construction process.

Hybrid Spatial Partitioning Schemes

There are other spatial partitioning algorithms that can be combined with BVHs to ensure smoothness in a scene in real time ray tracing. Often in real-time applications, hybrid solutions need to be created to ensure stability in Framerate.

5. Conclusion & Evaluation

In conclusion, this paper analyzes the impact of increasing the number of child nodes in a Bounding Volume Hierarchy, how it affects the Surface Area Heuristic Ratio, and the build times for a BVH.

The results show that increasing the maxBranchingFactor generally decreases the resources needed. The Surface Area Heuristic also generally decreases, with a few exceptions. Increasing the maxBranchingFactor predominantly affects the SAH ratio. The algorithm used to carry out this experiment was mainly a part of a high-performance Ray Tracing Kernel library called Intel Embree. The author modified it to add memory usage to the resulting output. The data also portray that a certain number of child nodes may work better for different scenes and computers because when the number of nodes is increased, the algorithm gets more complex. After all, traversing the BVH becomes more computationally intensive with each added node, increasing the algorithm's run time. Although the Build Time in the data opposes that, it is visible that when we increase the maxBranchingFactor past 4, the data starts to get inconsistent.

This research in the field of Computer Graphics is necessary because Ray Tracing uses acceleration structures like BVHs often. Usually, the more efficient BVH is, the smoother the Ray Intersection process will be, resulting in more accurate results and faster rendering times. Ray Tracing is very computationally expensive, and BVH structures help lower the computational requirements while increasing the efficiency. However, it is to be noted that

memory usage and traversal times depend mainly on the scene being used, the number of pixels, and the sampling rate.

There are some limitations of this essay. First, the experimentation conducted primarily focuses on varying the maxBranchingFactor parameter, although in a real world scenario, there are other factors and parameters that optimize the BVH construction process. Although this was beyond the scope of this essay, it is to be noted that factors such as

Another limitation was that the experiments were taken on only one type of Computer, although BVH construction is accurate on CPU more than a GPU, usually GPUs are used to render graphics because of their superior parallel processing power, which substitutes for their inaccuracy. This limits the broader application of this paper.

This paper helps show that research demonstrates the potential of improving the efficiency of BVH construction in Ray Tracing Applications by showing that increasing the maxBranchingFactor generally decreases resource usage. However, as discussed previously, it largely depends on many other factors excluding BVH.

Works Cited

Karras, T., & Aila, T. (n.d.). *Fast Parallel Construction of High-Quality Bounding Volume Hierarchies*. Research at NVIDIA. Retrieved December 28, 2023, from https://research.nvidia.com/sites/default/files/pubs/2013-07_Fast-Parallel-Construction/karras2013hpg_paper.pdf

Küçükkarakurt, F. (2021, December 28). *Geometry And Primitives In Game Development*. DEV Community. Retrieved August 21, 2023, from https://dev.to/fkkarakurt/geometry-and-primitives-in-game-development-1og

Md, S. (2023, October 21). ,. , - YouTube. Retrieved December 28, 2023, from https://www.sciencedirect.com/science/article/abs/pii/S0097849316301376

Nvidia Corporation. (2018). *Ray Tracing*. NVIDIA Developer. Retrieved June 27, 2023, from https://developer.nvidia.com/discover/ray-tracing

Pharr, M., Jakob, W., & Humphreys, G. (2023). *Physically Based Rendering, Fourth Edition: From Theory to Implementation*. MIT Press.

A Survey on Bounding Volume Hierarchies for Ray Tracing. (n.d.). Daniel Meister. Retrieved December 6, 2023, from https://meistdan.github.io/publications/bvh_star/paper.pdf

Visual Computing Systems. (2009, June 3). YouTube: Home. Retrieved December 15, 2023, from http://graphics.cs.cmu.edu/courses/15769/fall2016content/lectures/01_introhwreview/01_introhwreview_slides.pdf

Wiche, R. (2018, April 10). *How to create awesome accelerators: The Surface Area Heuristic*. Medium. Retrieved January 30, 2023, from

https://medium.com/@bromanz/how-to-create-awesome-accelerators-the-surface-area-heuristic-e14b5dec6160

Wodniok, D. (2016, November 7). *Construction of Bounding Volume Hierarchies with SAH Cost Approximation on Temporary Subtrees*. Retrieved December 28, 2023, from http://www.dominikwodniok.de/publications/Wodniok_CAG2017.pdf

Appendices

A1

Intel Embree is under License Apache-2.0, which allows the use of collaboration and modification on open source software (https://www.apache.org/licenses/LICENSE-2.0). Process of building Embree to ensure authenticity. All steps were done through the macOS terminal.

- Cloning the repository from GitHub (https://github.com/embree/embree.git)
- Navigating to the directory where the repository was cloned
- Navigating to (embree/tutorials/bvh_builder) and modifying the code in bvh_builder_device.cpp to add memory output
- Creating a build folder to store executable files
- Using the software Cmake (https://cmake.org/) to build Embree in the build repository
- Running the command ‘make’ to build the files so that they are executable

A2

Original code of bvh_builder_device.cpp

- GitHub link: (https://github.com/embree/embree/blob/master/tutorials/bvh_builder/bvh_builder_device.cpp)

A3

Sample of running the command ‘./embree_bvh_builder’ to execute bvh_builder. Here is a sample output of the modified code.

```
User@User-Air 5BREE % cd embree
User@User-Air embree % cd build
User@User- Air build % ./embree_bvh_builder
Low quality BVH build:
iteration 0: building BVH over 1000000 primitives, 57.4691ms, 17.4006 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 1: building BVH over 1000000 primitives, 24.1852ms, 41.3476 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 2: building BVH over 1000000 primitives, 22.054ms, 45.3433 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 3: building BVH over 1000000 primitives, 22.4121ms, 44.6188 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 4: building BVH over 1000000 primitives, 22.6839ms, 44.0842 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 5: building BVH over 1000000 primitives, 22.5792ms, 44.2886 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 6: building BVH over 1000000 primitives, 21.9052ms, 45.6513 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 7: building BVH over 1000000 primitives, 21.487ms, 46.5398 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 8: building BVH over 1000000 primitives, 22.141ms, 45.1651 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
iteration 9: building BVH over 1000000 primitives, 22.0878ms, 45.2738 Mprims/s, sah = 4.62233, memory used: 98444 KB [DONE]
Normal quality BVH build:
iteration 0: building BVH over 1000000 primitives, 99.412ms, 10.0592 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 1: building BVH over 1000000 primitives, 97.1558ms, 10.2927 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 2: building BVH over 1000000 primitives, 97.9311ms, 10.2113 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
```

```
iteration 3: building BVH over 1000000 primitives, 99.4711ms, 10.0532 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 4: building BVH over 1000000 primitives, 99.3719ms, 10.0632 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 5: building BVH over 1000000 primitives, 103.78ms, 9.63577 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 6: building BVH over 1000000 primitives, 116.66ms, 8.57193 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 7: building BVH over 1000000 primitives, 146.533ms, 6.8244 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 8: building BVH over 1000000 primitives, 117.867ms, 8.48414 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
iteration 9: building BVH over 1000000 primitives, 113.807ms, 8.78681 Mprims/s, sah = 14.1499, memory used: 182825 KB [DONE]
High quality BVH build:
iteration 0: building BVH over 1000000 primitives, 193.737ms, 5.16164 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 1: building BVH over 1000000 primitives, 184.681ms, 5.41474 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 2: building BVH over 1000000 primitives, 173.039ms, 5.77905 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 3: building BVH over 1000000 primitives, 171.064ms, 5.84576 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 4: building BVH over 1000000 primitives, 185.291ms, 5.39691 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 5: building BVH over 1000000 primitives, 198.328ms, 5.04215 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 6: building BVH over 1000000 primitives, 175.067ms, 5.7121 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 7: building BVH over 1000000 primitives, 176.677ms, 5.66005 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 8: building BVH over 1000000 primitives, 175.739ms, 5.69025 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
iteration 9: building BVH over 1000000 primitives, 175.391ms, 5.70155 Mprims/s, sah = 14.1522, memory used: 267207 KB [DONE]
```

A4

→ Modified code of bvh_builder_device.cpp (modified by the author). Here the arguments.maxBranchingFactor was changed to experiment with.

```
// Copyright 2009-2021 Intel Corporation
// SPDX-License-Identifier: Apache-2.0
#include "../common/tutorial/tutorial_device.h"
namespace embree
{
  RTCScene g_scene  = nullptr;
  ssize_t totalMemoryConsumed = 0; /*This Variable was added to track total memory consumed by the
algorithm*/
  /* This function is called by the builder to signal progress and to
   * report memory consumption. */
    bool memoryMonitor(void* userPtr, ssize_t bytes, bool post) {
       if (!post)
          totalMemoryConsumed += bytes;
       return true;
}
  bool buildProgress (void* userPtr, double f) {
    return true;
  }
  void splitPrimitive (const RTCBuildPrimitive* prim, unsigned int dim, float pos, RTCBounds* lprim,
RTCBounds* rprim, void* userPtr)
  {
    assert(dim < 3);
    assert(prim->geomID == 0);
```

```
    *(BBox3fa*) lprim = *(BBox3fa*) prim;
    *(BBox3fa*) rprim = *(BBox3fa*) prim;
    (&lprim->upper_x)[dim] = pos;
    (&rprim->lower_x)[dim] = pos;
  }
  struct Node
  {
    virtual float sah() = 0;
  };
  struct InnerNode : public Node
  {
    BBox3fa bounds[2];
    Node* children[2];
    InnerNode() {
      bounds[0] = bounds[1] = empty;
      children[0] = children[1] = nullptr;
    }
    float sah() {
      return 1.0f + (area(bounds[0])*children[0]->sah() +
area(bounds[1])*children[1]->sah())/area(merge(bounds[0],bounds[1]));
    }
    static void* create (RTCThreadLocalAllocator alloc, unsigned int numChildren, void* userPtr)
    {
      assert(numChildren == 2);
      void* ptr = rtcThreadLocalAlloc(alloc,sizeof(InnerNode),16);
      return (void*) new (ptr) InnerNode;
    }
    static void  setChildren (void* nodePtr, void** childPtr, unsigned int numChildren, void* userPtr)
    {
      assert(numChildren == 2);
      for (size_t i=0; i<2; i++)
        ((InnerNode*)nodePtr)->children[i] = (Node*) childPtr[i];
    }
    static void  setBounds (void* nodePtr, const RTCBounds** bounds, unsigned int numChildren, void*
userPtr)
    {
      assert(numChildren == 2);
      for (size_t i=0; i<2; i++)
        ((InnerNode*)nodePtr)->bounds[i] = *(const BBox3fa*) bounds[i];
    }
  };
  struct LeafNode : public Node
  {
    unsigned id;
    BBox3fa bounds;
    LeafNode (unsigned id, const BBox3fa& bounds)
      : id(id), bounds(bounds) {}
    float sah() {
      return 1.0f;
    }
```

```
    static void* create (RTCThreadLocalAllocator alloc, const RTCBuildPrimitive* prims, size_t numPrims, void*
userPtr)
    {
      assert(numPrims == 1);
      void* ptr = rtcThreadLocalAlloc(alloc,sizeof(LeafNode),16);
      return (void*) new (ptr) LeafNode(prims->primID,*(BBox3fa*)prims);
    }
  };
  void build(RTCBuildQuality quality, avector<RTCBuildPrimitive>& prims_i, char* cfg, size_t extraSpace = 0)
  {
    rtcSetDeviceMemoryMonitorFunction(g_device,memoryMonitor,nullptr);
    RTCBVH bvh = rtcNewBVH(g_device);
    avector<RTCBuildPrimitive> prims;
    prims.reserve(prims_i.size()+extraSpace);
    prims.resize(prims_i.size());
    /* settings for BVH build */
    RTCBuildArguments arguments = rtcDefaultBuildArguments();
    arguments.byteSize = sizeof(arguments);
    arguments.buildFlags = RTC_BUILD_FLAG_DYNAMIC;
    arguments.buildQuality = quality;
    arguments.maxBranchingFactor = 2;  /* maximum number of children nodes*
    arguments.maxDepth = 1024;
    arguments.sahBlockSize = 1;
    arguments.minLeafSize = 1;
    arguments.maxLeafSize = 1;
    arguments.traversalCost = 1.0f;
    arguments.intersectionCost = 1.0f;
    arguments.bvh = bvh;
    arguments.primitives = prims.data();
    arguments.primitiveCount = prims.size();
    arguments.primitiveArrayCapacity = prims.capacity();
    arguments.createNode = InnerNode::create;
    arguments.setNodeChildren = InnerNode::setChildren;
    arguments.setNodeBounds = InnerNode::setBounds;
    arguments.createLeaf = LeafNode::create;
    arguments.splitPrimitive = splitPrimitive;
    arguments.buildProgress = buildProgress;
    arguments.userPtr = nullptr;

    for (size_t i=0; i<10; i++)
    {
      /* we recreate the prims array here, as the builders modify this array */
      for (size_t j=0; j<prims.size(); j++) prims[j] = prims_i[j];
      std::cout << "iteration " << i << ": building BVH over " << prims.size() << " primitives, " << std::flush;
      double t0 = getSeconds();
      Node* root = (Node*) rtcBuildBVH(&arguments);
      double t1 = getSeconds();
      const float sah = root ? root->sah() : 0.0f;
      std::cout << 1000.0f*(t1-t0) << "ms, " << 1E-6*double(prims.size())/(t1-t0) << " Mprims/s, sah = " << sah;
      std::cout << ", memory used: " << totalMemoryConsumed / 1024 << " KB [DONE]" << std::endl;
```

```
    }

  rtcReleaseBVH(bvh);
}
/* called by the C++ code for initialization */
extern "C" void device_init (char* cfg)
{
  /* create random bounding boxes. 100000 was an arbitrary number */
  const size_t N = 1000000;
  const size_t extraSpace = 1000000;
  avector<RTCBuildPrimitive> prims;
  prims.resize(N);
  for (size_t i=0; i<N; i++)
  {
    const float x = float(drand48());
    const float y = float(drand48());
    const float z = float(drand48());
    const Vec3fa p = 1000.0f*Vec3fa(x,y,z);
    const BBox3fa b = BBox3fa(p,p+Vec3fa(1.0f));
    RTCBuildPrimitive prim;
    prim.lower_x = b.lower.x;
    prim.lower_y = b.lower.y;
    prim.lower_z = b.lower.z;
    prim.geomID = 0;
    prim.upper_x = b.upper.x;
    prim.upper_y = b.upper.y;
    prim.upper_z = b.upper.z;
    prim.primID = (unsigned) i;
    prims[i] = prim;
  }
  std::cout << "Low quality BVH build:" << std::endl;
  build(RTC_BUILD_QUALITY_LOW,prims,cfg);
  std::cout << "Normal quality BVH build:" << std::endl;
  build(RTC_BUILD_QUALITY_MEDIUM,prims,cfg);
  std::cout << "High quality BVH build:" << std::endl;
  build(RTC_BUILD_QUALITY_HIGH,prims,cfg,extraSpace);
}
void renderFrameStandard (int* pixels,
                          const unsigned int width,
                          const unsigned int height,
                          const float time,
                          const ISPCCamera& camera)
{
}

/* called by the C++ code to render */
extern "C" void device_render (int* pixels,
                               const int width,
                               const int height,
                               const float time,
```

```
                        const ISPCCamera& camera)
  {
  }
  /* called by the C++ code for cleanup */
  extern "C" void device_cleanup () {
  }
}
```