

Parallel BVH Construction for Real-Time Ray Tracing

Aaron Lemmon
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
lemmo031@morris.umn.edu

ABSTRACT

The rise in popularity of interactive graphical applications, such as video games, has motivated innovations in rendering 3-dimensional scenes in real time. The ray tracing technique is well-suited for generating realistic images of scenes that feature shadows, reflections, and refractions. Historically, ray tracing has been too slow for real-time applications since it is computationally intensive. However, ray tracing performance can be greatly improved by using the bounding volume hierarchy (BVH) acceleration data structure to store scene information for each frame. Researchers have strived to minimize the combined time of both constructing and using BVHs for ray tracing. This paper provides an overview of ray tracing with BVHs and presents a recently developed method for constructing them in parallel on a GPU.

Keywords

computer graphics, ray tracing, parallel computing, bounding volume hierarchies, Morton codes

1. BACKGROUND

In 3D computer graphics, objects are made up of a collection of *primitives*, which are usually simple geometric shapes like triangles. A 3D *scene* consists of all the primitives that construct it. Figure 1 depicts a scene with a dolphin that clearly shows the component triangles. In order to depict a scene on a display, the pixels of the display must be colored to create an image. A technique called *ray tracing* can be used to color the pixels in a way that can accurately portray shadows, reflections, and refractions in a scene [4]. Ray tracing achieves this by determining how light travels in a scene from the light sources, reflecting or refracting off objects, and meeting the viewer [6]. Figure 2 shows the high degree of photorealism that ray tracing can achieve.

Since many rays from a light source may not ultimately reach the viewer, it is more practical to start from the viewer and trace paths of light backward. Figure 3 shows that for every pixel on a display, a ray is traced from the viewpoint

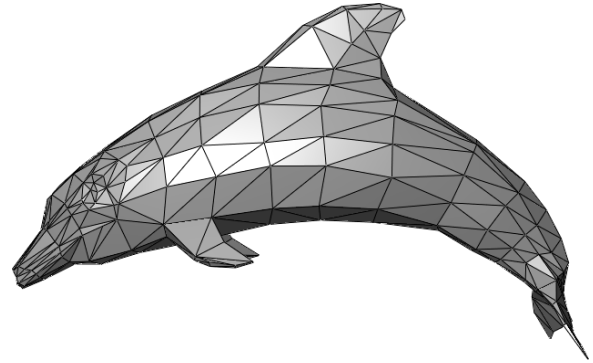


Figure 1: A simple scene containing a dolphin constructed from triangle primitives [7].

through the pixel and into the 3D scene. When a ray intersects with the first primitive in its path it recursively generates more rays in directions that will contribute to an appearance of reflections, refractions, or shadows [6]. While ray tracing can create highly realistic images, the process of tracing a large number of rays for a 3D scene with many primitives on a high resolution display can take a long time.

Ray tracing a single frame of a 3D scene can involve testing for intersections of billions of rays against millions of primitives [2, 5]. To speed up this process, acceleration data structures can be used to organize the primitives by their location so that only the primitives near a given ray need to be checked for intersection with that ray. Acceleration data structures usually take the form of a tree: the top node represents the entire 3D volume of the scene and the children of every node divide up the volume of their parent node into subsections. Although these acceleration data structures speed up ray intersection testing, the time it takes to build these trees can negatively impact performance [1]. This is especially apparent in scenes with moving objects since the acceleration data structures need to be rebuilt or updated to accurately reflect the new locations of objects [5].

The research discussed here addresses methods for building and maintaining acceleration data structures in a way that minimizes the combined time spent constructing the data structure and using it to test for ray intersections. In particular, parallel computing on the *graphics processing unit* (GPU) can decrease the time spent building an acceleration data structure. Efficient acceleration data structures allow for ray tracing scenes with motion in real time.

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

UMM CSci Senior Seminar Conference, April 2016 Morris, MN.



Figure 2: A ray traced scene featuring shadows on the wall and beneath the glasses, reflections on the glossy surfaces of the glasses, and refractions through the glass stems and ice cube [9].

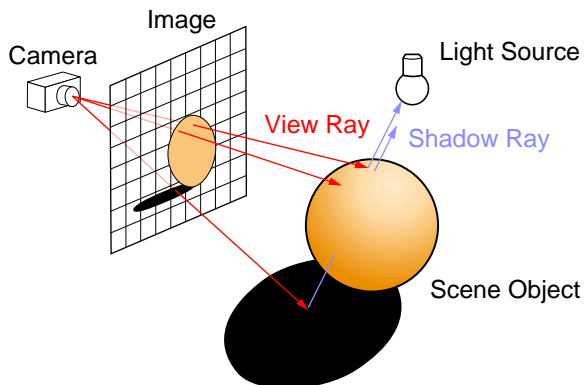


Figure 3: A ray is cast through every pixel of the image plane and tested for intersection with objects in the scene. When intersections occur, the angle of reflection is calculated and a new ray is sent out. Eventually, if a ray intersects with a light source, the color information propagates back to the pixel [2, 9].

2. ACCELERATION DATA STRUCTURES

If primitives of a scene are stored in a data structure that identifies their location in the scene, then a ray only needs to check for intersections with objects located in the parts of the scene the ray is passing through. This can drastically improve the performance of intersection testing for each ray [9].

A way to make intersection tests easier to calculate is to surround primitives with *bounding boxes*. A bounding box is a box that completely contains its contents as tightly as possible. A common approach is to use *axis-aligned bounding boxes* (AABBs), which are aligned with the axes of the scene as a whole. It is much simpler to test for ray intersections with AABBs than with the items they contain. If a ray does not intersect with an object's AABB, then it cannot intersect with the object itself. However, if it does intersect with the AABB, then a more costly check must be made to test for intersection with the contained object. Overall, using AABBs can reduce the cost of testing for intersections since a ray misses many more objects than it hits.

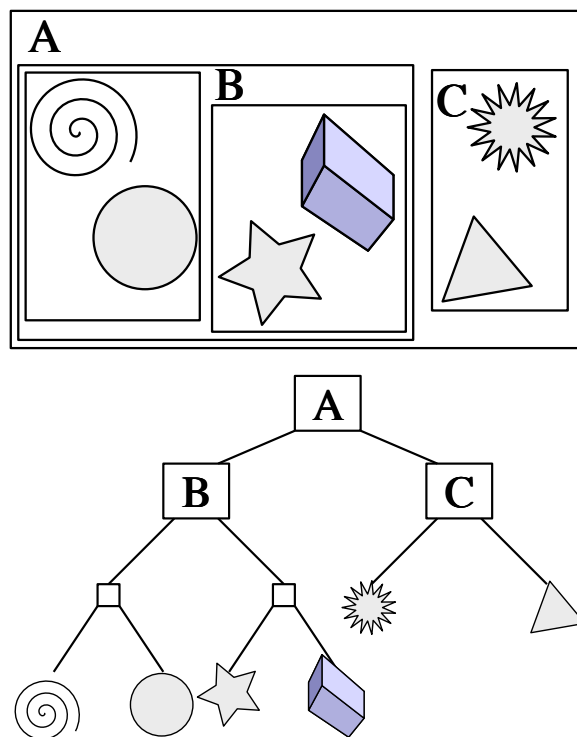


Figure 4: Top: A 2D scene with rectangles as bounding boxes. Bottom: One possible BVH configuration for the 2D scene. Note that each internal node has exactly two children and that objects that are close in the scene are also close in the BVH tree [8].

Bounding volume hierarchies (BVHs) extend the idea of AABBs to a binary tree data structure as shown in Figure 4. The root node of a BVH is an AABB that surrounds all the primitives in the scene. Every parent node has two child nodes, each of which is an AABB that surrounds its share of the primitives surrounded by the parent node. By continually separating volumes into smaller volumes, it becomes possible to group close primitives together [5]. The lower nodes on the tree give more precise location information than the higher nodes. The leaves of the tree are AABBs that surround a single primitive [8]. As opposed to other types of acceleration data structures, BVHs define the volumes by the objects they contain rather than splitting volumes and then determining which objects should go in each node [5].

Searching for a ray intersection with objects in a BVH occurs in a top-down manner. First, the ray is tested against the root of the tree to check if it even intersects with the scene. If it does, then both of the root's children are checked for intersection. If the ray misses a child node, then the entire subtree rooted at that node can be eliminated from the rest of the search, since the objects contained within the node will also not intersect. However, in the case that the ray intersects both children, the search must continue down both subtrees [2]. In the best-case scenario where that does not happen, the number of checks will be approximately the log of the number of total primitives. Ray tracing with a BVH structure can therefore greatly reduce the number of intersection checks performed per ray [8].

3. BVH CONSTRUCTION

In order to speed up the construction of BVHs for real-time ray tracing, Tero Karras [3] has developed a method based upon earlier work by Garanzha et al. [1] for constructing an entire BVH tree in parallel on a GPU. The method follows a series of four main steps. The method first assigns a value called a *Morton code* to each primitive based upon its location in the scene. The Morton codes are then sorted. Next, a *binary radix tree* (defined in Section 3.2.1) is constructed in parallel, which arranges close primitives near each other in the tree. The last step fits an AABB around the contents of each node in the binary radix tree in parallel to form the final BVH [3]. The next sections will cover each of these steps in more detail.

3.1 Morton Codes

The location of each primitive in the scene can be represented by the x , y , and z coordinates of the center of its AABB [4, 5]. The Morton code of a primitive combines the coordinates into a single value by interleaving the binary representations of the x , y , and z coordinates [1]. A Morton code has the form $X_0Y_0Z_0X_1Y_1Z_1\dots$ where the x coordinate is represented as the binary digits $X_0X_1X_2\dots$, and similarly for the y and z coordinates [3].

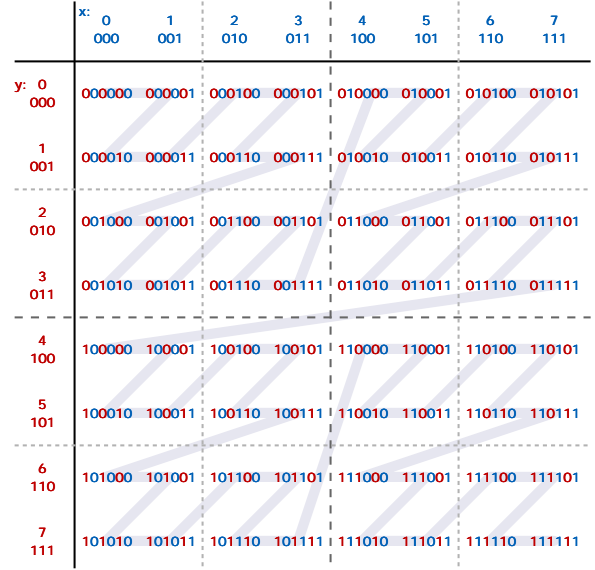
Figure 5 shows a two dimensional scene area with the Morton codes of each coordinate location. The lowest valued Morton code appears in the upper-left corner of the scene where both coordinates are zero. The zigzag pattern on the image shows the sequence of increasing Morton codes, which ultimately ends with the highest value in the lower-right corner. Every primitive in the scene is assigned a Morton code based on its location, and each code paired with a reference to its primitive is added to an array. Primitives that share a common location will be assigned the same Morton code specifying that location, while Morton codes for empty areas of the scene will be unused [3, 4].

Note that in Figure 5 all codes that start with a 0 bit are located in the upper half of the scene, and within that section all codes that have a 0 as the second bit are located on the left half of that section [1]. This property of Morton codes allows primitives that are near each other to have long *common prefixes* between their Morton codes, which is important for the binary radix tree construction. To arrange the Morton codes assigned to each primitive by their common prefixes, the array of codes is sorted without moving the actual primitives from their locations in the scene [4].

3.2 Binary Radix Tree Construction

3.2.1 Binary Radix Tree Fundamentals

The array of sorted Morton codes can be thought of as a set of n bit string keys k_0, \dots, k_{n-1} where n represents the number of primitives in the scene. A binary radix tree organizes the keys under a tree structure where the leaves are the keys and the internal nodes represent common binary prefixes of the keys. Figure 6 shows an example of the binary radix tree for a set of eight binary keys, which would be the Morton codes of primitives in a scene. Each key is a leaf node, and each internal node represents the longest common prefix shared by all the keys under it. These prefixes are always shorter than the full Morton codes, and longer prefixes represent a narrower set of locations in the scene. Every internal node always has exactly two children: the left child has the common prefix of the its parent followed by a 0 bit,



	x: 0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
y: 0	000000 000001	000100 000101	010000 010001	010100 010101	100000 100001	100100 100101	110000 110001	110100 110101
1	000010 000011	000110 000111	010010 010011	010110 010111	100010 100011	100110 100111	110010 110011	110110 110111
2	001000 001001	001100 001101	011000 011001	011100 011101	101000 101001	101100 101101	111000 111001	111100 111101
3	001010 001011	001110 001111	011010 011011	011110 011111	101010 101011	101110 101111	111010 111011	111110 111111
4	100000 100001	100100 100101	110000 110001	110100 110101	000000 000001	000100 000101	010000 010001	010100 010101
5	100010 100011	100110 100111	110010 110011	110110 110111	000010 000011	000110 000111	010010 010011	010110 010111
6	101000 101001	101100 101101	111000 111001	111100 111101	001000 001001	001100 001101	011000 011001	011100 011101
7	101010 101011	101110 101111	111010 111011	111110 111111	001010 001011	001110 001111	011010 011011	011110 011111

Figure 5: Morton codes for each location in a 2D scene. Notice that each Morton code is made by interleaving the bits of the x (blue) and y (red) coordinates for its location [10].

and the right child has the same common prefix but followed by a 1 bit [3]. As an example, the left child of the root in Figure 6 has the prefix “00”, so every key under its left child begins with “000” and every key under its right child begins with “001”. Since every internal node has exactly two children, a binary radix tree with n leaf nodes will always have exactly $n - 1$ internal nodes [5].

Every key in a binary radix tree must be unique, but duplicate Morton codes could exist if the primitives are extremely close. To ensure uniqueness, the binary representation of each key’s index in the array can be concatenated onto the end of the key. These concatenations do not need to be stored, but can be performed as needed when comparing identical keys [3]. For example, if two keys both have the Morton code “010” and they are stored at indexes (in binary) 000 and 001, then the index concatenation will result in the unique values “010000” and “010001”.

3.2.2 Binary Radix Tree Properties

In order to create every internal node of the binary radix tree construction in parallel, it must be possible to determine three properties about a node. These properties include the range of keys that a node covers, the length of the longest common prefix of those keys, and what the node’s children are. Additionally, these must be determined without depending on the work done in any other internal nodes since all the nodes will be working in parallel. This section will cover the three important properties of an internal node and Sections 3.2.3 and 3.2.4 will discuss the binary radix tree construction.

The keys covered by an internal node can be represented as a linear range $[i, j]$. Using Figure 6 as an example, the root node covers keys 0 through 7, so it can be represented as the range $[0, 7]$. Its left child can be represented as $[0, 3]$ and its right child as $[4, 7]$.

The length of the longest common prefix between two keys

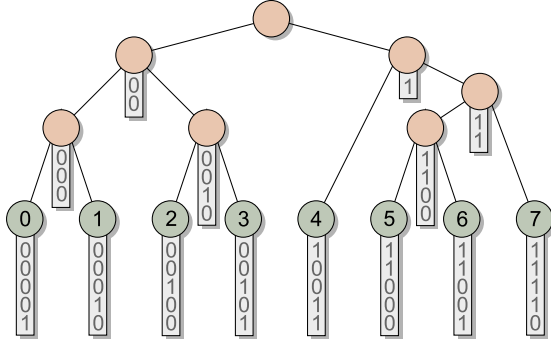


Figure 6: An ordered binary radix tree. There are eight leaf nodes containing 5-bit keys which appear in lexicographical order. Each internal node covers a linear range of keys with a common prefix, which is displayed directly beneath the node. The range of keys covered by an internal node is partitioned into two ranges according to their first differing bit; the two subranges are represented by the two children of the internal node [3].

k_i and k_j is denoted by $\delta(i, j)$. For example, the longest common prefix between key 0 and key 3 in Figure 6 is “00”, which has a length of two digits. So $\delta(0, 3)$ is 2.

The prefix for each node can be determined by solely inspecting the first and last key in its key range. This is because all keys between the first and last key will also share the same prefix since all keys are in lexicographical order [3]. As an example, the left child of the root in Figure 6 covers keys 0 through 3. By only looking at keys 0 and 3, it can be determined that they share the prefix “00”. All keys between 0 and 3 will also have the prefix “00”, otherwise they would not fall in that range.

An internal node partitions its range of keys into two subranges for its children according to the first differing bit among the keys. For example, the left child of the root in Figure 6 covers keys 0 through 3 with a common prefix of “00”. This node will divide its keys among its children based on the value of the third bit of the keys, since the third bit comes directly after the common prefix. All keys in the range with 0 as the third bit will belong under the left child and the keys with 1 as the third bit will belong under the right child. The index of the last key where the differing bit is 0 is called the *split position* for the node, and is denoted by γ . If the range of keys covered by a node is $[i, j]$, then the split position can be anything from i to $j - 1$. The split position cannot occur at j , since the differing bit must be a 1 for key j . The split position for the left child of the root in Figure 6 is 1 because key 1 is the last key in the range $[0, 3]$ with the prefix “000”, and the next key must have the prefix “001”. The subrange $[i, \gamma]$ represents the range of keys covered by the left child and the subrange $[\gamma + 1, j]$ represents the range covered by the right child [3].

For an internal node with range $[i, j]$, the equality $\delta(\gamma, \gamma + 1) = \delta(i, j)$ always holds true and is important for finding the split position. In fact, k_δ and $k_{\delta+1}$ is the only pair of adjacent keys in the range where the prefix length of the pair is equal to the prefix length of the range. Pairs of adjacent keys to the left of the split position will have a longer common prefix than the range since they all have 0

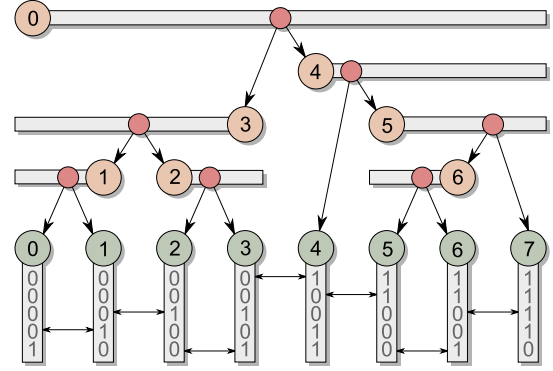


Figure 7: This figure shows the storage locations of nodes for the binary radix tree displayed in Figure 6. The internal nodes have been assigned indices from 0 to 6 and are lined up with leaf nodes with the same indices. The range of keys covered by each internal node is shown as a horizontal bar, and a red circle lies immediately after each node’s split position [3].

in the differing bit position. Adjacent keys to the right of the split position are analogous, but with all having a 1 in the differing bit position.

3.2.3 Setup for Binary Radix Tree Construction

It is possible to construct a binary radix tree by starting with the root node, finding the first differing bit in the keys, creating the child nodes, then handling each child recursively. However, this is not parallel because a node cannot be created until all of its ancestors have been created. Parallel construction can be achieved by creating a connection between node indices and keys through a particular tree setup. This is done by assigning indices to internal nodes in a way that enables finding their children without depending on that node’s ancestors being complete [3].

Figure 7 presents a visual representation of how the nodes of the binary radix tree from Figure 6 will be stored. The leaf nodes and internal nodes are stored in two separate arrays, L and I , respectively. The root will always be located at position I_0 . The children of internal nodes are placed at an index according to the internal node’s split position. The left child is located at L_γ if it is a leaf, or at I_γ if it covers more than one key. Similarly, the right child is located at $L_{\gamma+1}$ or $I_{\gamma+1}$ [3]. For example, the split position of the root in Figure 7 is at index 3, so its left child is stored in I_3 and its right child is stored in I_4 .

It is important to note that the index of every internal node is equal to the index of either its first or last key. The index of a node will be equal the index of its first key if the node is a right child of its parent. Figure 7 shows that internal node 4 is a right child of the root, that its index coincides with its first key (key 4), and that its range extends rightward. Conversely, internal node 3 is a left child of the root, its index coincides with its last key (key 3), and its range extends leftward. More generally, if a node covers the range $[i, j]$, then its left child will be located at γ , which is the end of its range $[i, \gamma]$. The right child will be located at $\gamma + 1$, which is the beginning of its range $[\gamma + 1, j]$ [3].

3.2.4 Binary Radix Tree Construction Algorithm

```

1: for each internal node with index  $i \in [0, n-2]$  in parallel
2:   // Determine direction of the range (+1 or -1)
3:    $d \leftarrow \text{sign}(\delta(i, i+1) - \delta(i, i-1))$ 
4:   // Compute upper bound for the length of the range
5:    $\delta_{\min} \leftarrow \delta(i, i-d)$ 
6:    $l_{\max} \leftarrow 2$ 
7:   while  $\delta(i, i+l_{\max} \cdot d) > \delta_{\min}$  do
8:      $l_{\max} \leftarrow l_{\max} \cdot 2$ 
9:   // Find the other end using binary search
10:   $l \leftarrow 0$ 
11:  for  $t \leftarrow \{l_{\max}/2, l_{\max}/4, \dots, 1\}$  do
12:    if  $\delta(i, i+(l+t) \cdot d) > \delta_{\min}$  then
13:       $l \leftarrow l+t$ 
14:   $j \leftarrow i+l \cdot d$ 
15:  // Find the split position using binary search
16:   $\delta_{\text{node}} \leftarrow \delta(i, j)$ 
17:   $s \leftarrow 0$ 
18:  for  $t \leftarrow \{\lceil l/2 \rceil, \lceil l/4 \rceil, \dots, 1\}$  do
19:    if  $\delta(i, i+(s+t) \cdot d) > \delta_{\text{node}}$  then
20:       $s \leftarrow s+t$ 
21:   $\gamma \leftarrow i+s \cdot d + \min(d, 0)$ 
22:  // Output child pointers
23:  if  $\min(i, j) = \gamma$  then  $\text{left} \leftarrow L_{\gamma}$  else  $\text{left} \leftarrow L_{\gamma}$ 
24:  if  $\max(i, j) = \gamma+1$  then  $\text{right} \leftarrow L_{\gamma+1}$  else  $\text{right} \leftarrow L_{\gamma+1}$ 
25:   $I_i \leftarrow (\text{left}, \text{right})$ 
26: end for

```

Figure 8: Pseudocode for binary radix tree construction. Note that if an out-of-bounds argument is passed to $\delta()$, it is defined to return -1 [3].

The goal of the binary radix tree construction algorithm is to determine the range $[i, j]$ of keys covered by each internal node, as well as the indices of its children. One end of the range is given by the internal node's index, and the other end of the range can be found by examining the surrounding keys. The indices of the children can be found by locating the split position. The previous setup allows each internal node to be processed independently and in parallel with the other internal nodes [3]. This section will explain the construction process using the pseudocode in Figure 8 as a reference.

Consider processing an internal node I_i , which is in the i th position of the array I . First, the direction the range extends (right or left) is calculated in line 3 of Figure 8. If d becomes +1 then the range extends rightward; if it becomes -1 then the range extends leftward [3]. For example consider $i = 2$, which covers keys 2 and 3 as shown in Figure 7. In this example, $\delta(2, 3) = 4$ while $\delta(2, 1) = 2$, so the range of node 2 extends to the right since it shared a longer common prefix with its right neighbor.

The next task is to determine j (handled in lines 5-14 in the pseudocode), which represents the index of the other end of the range $[i, j]$. For the example, j is 3 since the range of the node at index 2 extends from index 2 to index 3. Every internal node covers at least two keys, so k_i and k_{i+d} must belong to I_i . The other neighboring key k_{i-d} must belong to I_{i-d} . The two keys k_i and k_{i+d} share a common prefix that is different and longer than the prefix between k_i and k_{i-d} . Let δ_{\min} represent the value of $\delta(i, i-d)$. In the example, δ_{\min} equals 2, with the common prefix being "00". The value of δ_{\min} is equivalent to the length of the prefix represented

by the parent of the node currently under consideration. This fact is used to determine the index of j . For any k_m belonging to I_i , the inequality $\delta(i, m) > \delta_{\min}$ always holds true. Therefore, δ_{\min} gives a lower bound for the length of the prefix for the section of keys covered by I_i . So the index j marking the other end of the range can be found by searching for the largest l that satisfies $\delta(i, i+l \cdot d) > \delta_{\min}$. Then j will be the index $i+l \cdot d$ [3].

Finding j is achieved by first finding a power-of-two exclusive upper bound for l , denoted as l_{\max} (done in lines 6-8 of the pseudocode). As shown in Figure 8, l_{\max} begins at 2 and is doubled until it becomes the upper bound for l . For example, for I_5 , l_{\max} is doubled until it reaches 4. We can see from Figure 7 that l is 2 for node 5 since it covers the keys $[5, 5+2]$. Therefore, $l_{\max} = 4$ is the correct exclusive power-of-two upper bound for node 5 [3].

Once the upper bound l_{\max} is determined, l can be found by using binary search in the range $[0, l_{\max} - 1]$ (performed in lines 10-13). For example, for node 5, with $\delta_{\min} = 1$ and $l_{\max} = 4$, the loop goes through the range $\{2, 1\}$. l starts at 0 and is incremented by 2 in the first iteration, but not incremented by 1 in the second iteration since the condition to increment does not hold. This gives a final l value of 2. Now j can be determined by the formula $j = i+l \cdot d$, which is 7 for node 5. This is correct since node 5 covers the range $[5, 7]$.

Now that the range of keys for the node has been determined, the length of the common prefix of those keys, denoted by δ_{node} , is calculated as $\delta(i, j)$ on line 16 of Figure 8. The next step of the algorithm is to determine where the range of keys should split for the left and right children. Recall from the last paragraph of Section 3.2.2 that $\delta(\gamma, \gamma+1) = \delta(i, j)$, which is the value of δ_{node} . This equality occurs since the first differing bit in the range of keys toggles exactly between γ and $\gamma+1$. To find this split location, the next goal is to determine the largest s in the range $[0, l-1]$ that satisfies $\delta(i, i+s \cdot d) > \delta_{\text{node}}$. The index $i+s \cdot d$ will be the furthest index from i in the node's key range where the first differing bit is the same as that of key i . If the range extends rightward from i then $i+s \cdot d$ will be γ . Conversely, if the range extends to the left from i then $i+s \cdot d$ will be $\gamma+1$. The value of s can be found using the binary search shown on lines 17-20 of the pseudocode, which works similarly to the previous binary search [3].

Now that s is known, γ is determined by $i+s \cdot d + \min(d, 0)$, which appears on line 21 of the pseudocode. The addition by $\min(d, 0)$ serves to place γ to the left of the split if the range was extending to the left. In that case, $i+s \cdot d$ yields $\gamma+1$ instead of γ , since the search was scanning from right to left [3]. Consider the full formula for node 3, where $s = 1$ and $d = -1$, indicating that the range extends left. The formula gives $\gamma = 3 + (1 \cdot -1) + (-1) = 1$, which matches the index of the left child for node 3 in Figure 7.

Lastly, the node stores the indexes of its children as shown on lines 23-25 of Figure 8. These assignment statements ensure that children covering only one key are represented as leaf nodes, while children covering multiple keys are represented as internal nodes.

Overall, when running this algorithm in parallel on the GPU, each thread is responsible for only one internal node. The balance of the tree depends solely on the locations of primitives in the scene. Scenes with an even distribution of primitives throughout the space will generate well-balanced trees.

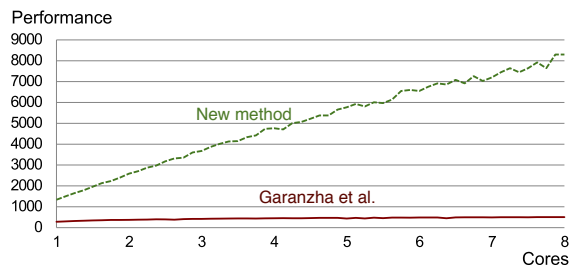


Figure 9: A comparison of two algorithms. The y axis represents millions of primitives per second and the x axis is the number of parallel cores. [3].

3.2.5 Time Complexity

For a node that covers q keys, each of the three loops above executes at most $\lceil \log_2 q \rceil$ iterations. Since $q \leq n$ for all $n - 1$ internal nodes, the worst-case time complexity for constructing the entire tree is $\mathcal{O}(n \log n)$. The worst case occurs when the height of the tree grows proportional to n , but this is unlikely since the height of the tree is limited to the length of the keys [3].

3.3 Fitting AABBs

To turn the binary radix tree into a BVH, the contents of each node are surrounded with an AABB. The leaves are already done since each primitive has its own AABB as discussed in Section 2. Each internal node can construct an AABB by surrounding its two children's AABBs in a box in a manner similar to Figure 4. Since the dimensions of each internal node's AABB depends on the dimensions of each child's AABB, it must be done in a bottom-up fashion. To do this in parallel, each thread should start from a leaf node and travel up the tree from there. However, each internal node has two children and it should not be processed twice. Therefore, each internal node keeps an atomic count of how many threads have visited it. The first thread that reaches an internal node terminates immediately, while the second thread gets to process the node [3].

3.4 Results

As shown in Figure 9, the performance of the algorithm scales very well as the number of cores increases. The execution time is inversely proportional to the number of cores. This is an excellent quality because it means that the performance can be roughly doubled by doubling the number of cores assigned to the task [3]. The figure also shows a comparison between this algorithm and the best previously known algorithm by Garanzha et al. As the number of cores increases, this algorithm greatly surpasses the performance of the comparison method, even though the comparison method has elements of parallelism [1].

4. CONCLUSION

The ray tracing technique is exceptional at producing high quality images of 3D scenes. The simulation of actual light propagation results in the shadows, reflections, and refractions which make images appear photorealistic. By using both the parallel BVH construction algorithm presented in this paper and hardware with multiple cores, real-time ray tracing becomes significantly faster. As the number of cores

on modern hardware continues to increase, this algorithm will become even more impactful.

While this algorithm is a significant improvement over previous work, it is not without flaws. Scenes with poorly distributed primitives will generate BVHs that are not well-balanced. This is because the algorithm uses absolute locations of the primitives in the scene, rather than their positions relative to each other. An additional drawback to this algorithm is that it creates a new BVH for each frame without using any information from the previous frame. In sequences of frames where primitives move very little, it would be beneficial to simply adjust the previous frame's BVH, rather than constructing an entirely new one. Further research is needed to overcome these drawbacks. Overall, this research presents remarkable progress in the field of ray tracing.

Acknowledgments

Thanks to Nic McPhee, Elena Machkasova, K.K. Lamberty, Max Magnuson, and Emma Sax for their time, feedback, and constructive comments.

5. REFERENCES

- [1] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 59–64, New York, NY, USA, 2011. ACM.
- [2] C. Gribble, J. Fisher, D. Eby, E. Quigley, and G. Ludwig. Ray tracing visualization toolkit. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 71–78, New York, NY, USA, 2012. ACM.
- [3] T. Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.
- [4] T. Viitanen, M. Koskela, P. Jääskeläinen, H. Kultala, and J. Takala. Mergetree: A HLBVH constructor for mobile systems. In *SIGGRAPH Asia 2015 Technical Briefs*, SA '15, pages 12:1–12:4, New York, NY, USA, 2015. ACM.
- [5] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.
- [6] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.
- [7] Wikipedia. Triangle mesh — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 10-April-2016].
- [8] Wikipedia. Bounding volume hierarchy — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 10-April-2016].
- [9] Wikipedia. Ray tracing (graphics) — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 10-April-2016].
- [10] Wikipedia. Z-order curve — Wikipedia, The Free Encyclopedia, 2016. [Online; accessed 10-April-2016].