A Bucket LBVH Construction and Traversal Algorithm for Volumetric Sparse Data

I.B. Fernandes and M. Walter
Institute of Informatics - PPGC
Universidade Federal do Rio Grande do Sul - Brasil

Fig. 1: Datasets used as tests cases rendered in our in-house Narval Engine. From left to right: Harvard Dragon, Fireball, Explosion, Disney Cloud and Bunny cloud.

***Abstract*—Many volumetric rendering algorithms use spatial 3D grids as the underlying data structure. Efficient representation, construction, and traversal of these grids are essential to achieve real-time performance, particularly for time-varying data such as in fluid simulations. In this paper, we present improvements on algorithms for building and traversing Bounding Volume Hierarchies (BVH) designed for sparse volumes. Our main insight was to simplify data layout representation by grouping voxels in buckets, preserving their spatiality using Morton codes, instead of using bricks, as current solutions use. Our solution does not use pointers nor stacks, allowing for its usage directly on computing shaders and provides, on average, 9.3x improvement in construction speed, compared with state-of-the-art approaches for Linear Bounding Volume Hierarchies (LBVH).**

I. Introduction

Volumetric rendering is a field in computer graphics focused on techniques to render spatial data often not defined by geometric primitives, such as medical computed tomography scans and participating media. These data are typically stored in 3D uniform grids. Accurate volumetric rendering has been a challenge to do in real-time, due to the memory footprint needed to store increasingly larger data sets and the required time to render them. Most real-time systems in the game industry using volumetric rendering are constrained to small resolution grid sizes to fit both speed and memory constraints [1]. Specialized data structures can ease the problem, and a BVH built-in linear time and implemented in GPU is currently the state-of-the-art for these tasks [2].

In this work, we present a stackless and pointerless algorithm to build and traverse LBVHs using buckets. Without stacks nor pointers, we can quickly transfer our algorithms to shaders, which are the most common way for real-time rendering on the game industry. Since we do not use bricks and directly store voxels, we avoid the potential overhead that comes with vastly sparse volumes, where if a brick has only a single non-empty voxel, we still need to store all other voxels within this brick.

Our proposal does not present the binary tree limitation, being easily adapted to a quadtree, octree, or any other power of two n-dimensional tree by changing how we calculate the parent-children relationship between nodes. Different tree structures work differently for each data distribution, whereas in some cases, a shorter tree performs better than a deeper tree. We also aimed to implement a more straightforward algorithm linearized in memory that performs better for real-time rendering usage.

II. Related work

As soon as ray tracing was introduced [3], the need for Bounding Volume Hierarchies (BVH) was clear [4]. A bit later, with the availability of computed tomography (CT) scans, marching cubes [5] also used spatial data structures to store volume data, which are converted into 2D images. As rays cross the space to compute intersections, we need efficient ways to traverse it when searching for geometric or volumetric primitives. With a hierarchical data structure segmenting regions by existing objects, we can quickly remove large portions of the space. A vast amount of work improved on these initial ideas.

Efficient support for dynamic data sets appeared around 2006, with a focus on rebuilding BVHs [6]. Almost at the same time, we start to see GPU implementations for dynamic BVHs. Torres, Martin and Gavilanes [7] addressed the lack of recursion on GPUs and proposed a *roped* version of BVH using CUDA. Ropes are pointers to the next node in a preorder tree traversal.

To be amenable to run in GPUs, Lauterbach and colleagues [8] introduced the idea of a *Linear* BVH (LBVH), by first traversing the input primitives according to a Morton space-filling curve and thus "linearizing" the primitives. The Morton codes are then sorted and processed in this order. Their work achieved substantial gains in the construction step, with a positive net effect when both construction and traversal are taken into account, but only for surface primitives.

Pantaleoni and Luebke presented HLBVH, a hierarchical approach for LBHV [9]. By building a hierarchy using Morton codes [10], thus preserving spatial mesh coherence, they reported 2-3x speedup over a reference implementation of the LBVH [8].

Murguia and colleagues improved on the idea of LBVH by presenting a stackless approach called SLBVH. Their first contribution presented a CPU-version later extended for GPUs. Both approaches improved construction times while preserving traversal and were used for surface primitives only, such as triangles. Their technique presented in [11] benefits from a fast way to find left and right children of a parent node in a binary tree, reducing memory accesses. Their CPU implementation also introduced a stackless and pointerless approach for building a heap as an LBVH. Their second paper [12] was mainly concerned with introducing a GPU implementation of their previous work, together with a speeding up scheme where traversals could start at the last visited node instead of the root using a "bit-trail" approach. Another variant presented by Hapala and colleagues [13], used a parent pointer within each node of a BVH, plus simple state logic to infer which node to traverse next.

Together with these previous approaches, we start to see specialized solutions for volume rendering, where the primitives are not surfaces, but voxels. In the work done by Fogal and colleagues [14], the authors explore several combinations of previously studied strategies and propose a new renderer. Particularly interesting is their analysis of brick sizes for some datasets. A brick is a uniform box-shaped subdivision of the volume space. Bricking allows us to load and render as required, instead of using the whole volume [15]. In 2014, Beyer, Hadwiger, and Pfister presented a report on GPU-Based Large-Scale Volume Visualization [15]. Although six years old now, it provides a pedagogical text for many concepts in this research line.

More recently, Zellmann and his group managed to expand on the original LBVH idea [8], adapting it to deal with direct volume rendering with sparse volumes [2], since the LBVH formerly assumed only triangles as primitives. Instead of sorting triangles, they used bricks of fixed size. One of their limitations is that their algorithm in CUDA could not be ported to shaders, since they used pointers and stacks for their implementation. In the next section, we detail Zellman's approach since we used it as our departure point.

Finally, Stoter and colleagues presented yet another variant for LBVH called Octree LBVH (OLBVH) [16]. It is a specialized octree-based data structure targeted for unstructured volumetric meshes, such as the ones that appear in finite element simulations and 3D printing tasks.

III. Bucket LBVH Algorithm

Zellman et al. [2] approach to a LBVH first decomposes the 3D uniform grid into bricks of fixed resolution 8^3. On the next step, they run a CUDA kernel where each thread is responsible for deciding whether a voxel is visible or not for the current transfer function. Each thread is made responsible for one brick and atomically votes if the brick is visible or not. Then it sets a bit flag accordingly, followed by a compaction operation to remove empty bricks. For each non-empty brick, they assign a 30-bit 3D Morton code that is used to sort them calling functions from the Thrust library [17]. The order of sorted Morton codes implicitly defines a spatial hierarchy for all bricks, spatially splitting children nodes in the middle. Split positions can be read from the bit codes of the Morton indices and computed using Karras' algorithm [18]. In the last step, a CUDA kernel traverses the tree hierarchy on a bottom-up fashion while assembling the Axis Aligned Bounding Box (AABB) of each node.

Morton codes allow sorting points alongside a space-filling curve that preserves data locality. Figure 2 shows how these codes are distributed in a 3D space. For example, the Morton encoded coordinate (31, 9, 11) is equal to 20271, as shown below.

$$
\begin{aligned}
(31, 9, 11) &= (11111_2, 01001_2, 01011_2) \\
\mathbf{100}0000000000000_2 &= (\mathbf{1}1111_2, \mathbf{0}1001_2, \mathbf{0}1011_2) \\
100\mathbf{111}000000000_2 &= (1\mathbf{1}111_2, 0\mathbf{1}001_2, 0\mathbf{1}011_2) \\
100111\mathbf{100}000000_2 &= (11\mathbf{1}11_2, 01\mathbf{0}01_2, 01\mathbf{0}11_2) \\
100111100\mathbf{101}000_2 &= (111\mathbf{1}1_2, 0100\mathbf{1}_2, 0101\mathbf{1}_2) \\
100111100101\mathbf{111}_2 &= (1111\mathbf{1}_2, 0100\mathbf{1}_2, 0101\mathbf{1}_2) \\
100111100101111_2 &= 20271_{10}
\end{aligned}
$$

First, convert each axis coordinate into its binary form. For each most significant bit of each coordinate axis in the indices n_x, n_y, and n_z, pack them in the ZYX order and put them into the most significant "pack" n of the Morton code.

In our CUDA implementation, we do not code bricks, but each AABB coordinate representing a voxel. Our approach consists of encoding each non-empty voxel using 30 bit Morton codes, 10 bits for each axis, and sort them using the Thrust library, as illustrated in Figure 3, steps 1 and 2. On step 3, we calculate the minimum amount of buckets needed and fill them with their respective Morton codes. With all buckets filled, we proceed to shift all non-empty buckets to the left and map each bucket to a tree leaf, as demonstrated in steps 4, 5, and 6. The final step is to propagate each node's AABB along the tree in a bottom-up fashion. Each tree node comprises of two 32 bit integers representing its AABB minimum and maximum coordinates encoded as a simple 10-bit shift of each axis in the ZYX order, as shown in the example below. Using 32 bits integers, the system is limited to a grid with a maximum resolution of 1024^3, or 2^{10} per axis. It can be trivially extended to use 64 bits integers, expanding the maximum resolution to $2,097,152^3$, or 2^{21} per axis.

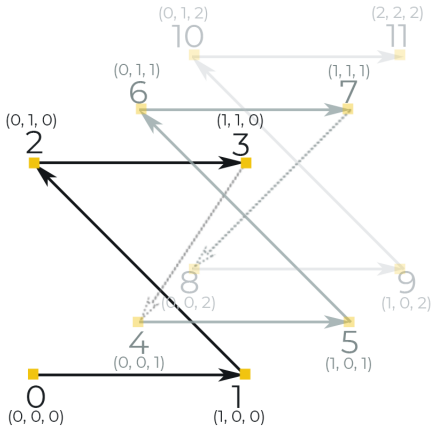


Fig. 2: Morton code filling curve presented in a 3D space where each yellow dot illustrates a voxel alongside its 3D space coordinates and corresponding Morton encoding.

$$
\begin{aligned}
&(1023, 829, 560) = \\
&\quad 560 \ll 20 = 00\mathbf{1100110001}00000000000000000000 \\
&\quad 829 \ll 10 = 00110011000\mathbf{1100111101}0000000000 \\
&\quad 1023 \ll 0 = 00110011000111001111101\mathbf{1111111111}
\end{aligned}
$$

Each tree leaf also has a corresponding value stored in an offset array that encodes both the number of voxels inside its bucket and the starting index at which these values begin in the sorted Morton codes array. In Section III-A we discuss in details how we construct the tree and in Section III-B we discuss its traversal.

A. Construction

The Stackless and Pointerless Bucket Linear Bounding Volume Hierarchy (BLBVH) construction starts by generating a Morton code [10] for each non-empty voxel in the grid. Since volumes stored as 3D uniform grids contain voxels with fixed-sized AABBs, we deliberately chose the minimum corner of each voxel to be encoded by a Morton code. It could be any point as long as the same procedure is used for all voxels within the grid. Given the minimum corner, it is trivial to calculate the maximum corner that forms each AABB by adding one or any other size given to a voxel, thus eliminating the need to store both corners in memory.

Once the array of Morton codes is computed, we proceed to sort them. Before generating leaf nodes, we create an array of offsets, used to calculate at which index the elements of each bucket start on the sorted Morton codes array, and how many voxels are inside each bucket. Given the highest Morton code value h and bucket size b, the minimum amount of offsets and buckets necessary to store all Morton codes is given by $\frac{h}{b}$. Each offset is filled with the number of corresponding Morton codes that fall within its range. Next, we do a parallel inclusive scan on all offsets, resulting in each element being a cumulative sum of all previous values in the array.

We generate tree leaves using the previously calculated array of Morton codes and offsets. The number of elements in each bucket is given by $offsets[i] - offsets[i-1]$, where $offsets[i]$ corresponds to an index in the Morton codes array at which this buckets elements start. We calculate the AABB of each bucket by getting the minimum and maximum of all the voxels it contains. For all buckets that are still empty, i.e., $offsets[i]$ - $offsets[i-1] = 0$, we set a flag on the first bit of the Morton code on both node's min and max, indicating that this node is empty, thus saving the need for an intersection test when ray casting.

For every tree level, we launch a kernel that calculates its current node min and max based on its children AABBs. The final memory layout of the tree is an array of mins and maxes representing the AABB of each node. The whole process is illustrated in Figure 3 and written down as pseudo-code in Algorithm 1.

Algorithm 1: Bucket LBVH binary tree construction

```
procedure treeConstruction():
    foreach Voxel v in Grid do
        if v is non-empty then
            MortonCodes[i] = encodeMorton(v)
        end
    end
    sort(MortonCodes)
    findAndCompactEmptyBuckets()
    depth = log2(quantity of non-empty buckets) + 1

    //Each bucket maps to a leaf
    foreach bucket b do
        if b is empty then
            setEmptyBitFlag(node)
        else
            tempAABB
            foreach Voxel v inside b do
                tempAABB = minmax(tempAABB, v)
            end
            simpleEncode(node, tempAABB)
        end
    end

    foreach level l from [0, depth - 2] do
        foreach node n in this level do
            if n.leftChild and n.rightChild is empty then
                setEmptyBitFlag(n)
            else
                AABB = minmax(n.leftChild, n.rightChild)
                simpleEncode(n, AABB)
            end
        end
    end
```

B. Traversal

Traversal occurs similarly to the classical Depth First Search (DFS) on LBVHs. For every visited node, starting from the root, we first check for the empty bit flag. If it is empty, we do not check for any intersection and move on to the next node to be visited. If the current node is not empty, and an intersection test succeeds, we test its left child. Since we want a flexible way to handle any power of 2 trees, we use a more generic

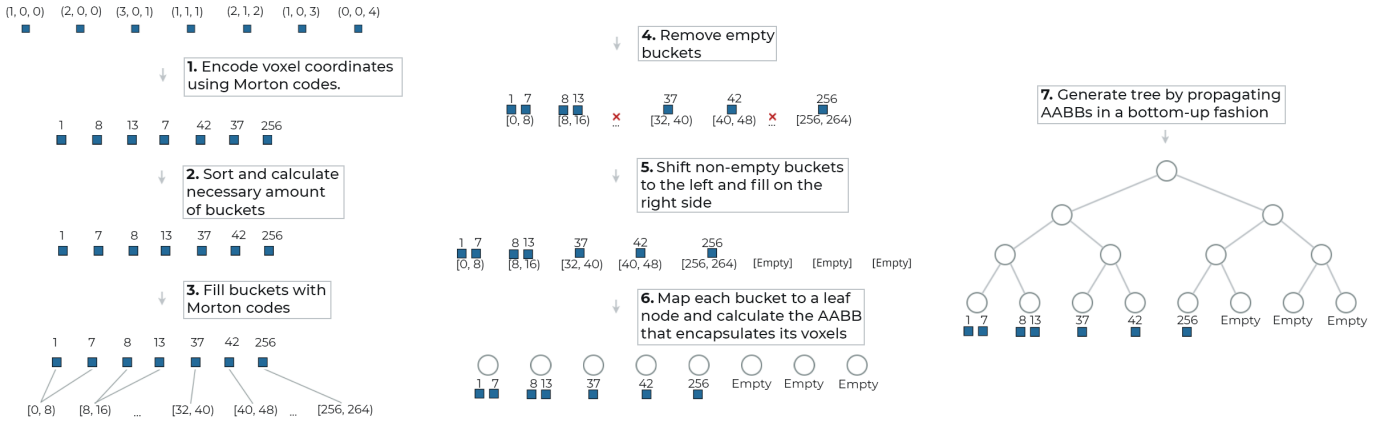


Fig. 3: Tree construction step by step. In step 1 the coordinates of each voxel are encoded using Morton Codes. In step 2 the voxels are sorted based on their Morton codes and the minimum amount of buckets is calculated. In step 3 each bucket is filled with the voxels that fall within its range. In step 4 all buckets that did not receive a voxel, i.e. are empty, are removed from the list. In step 5 all non-empty buckets are shifted to the left and the remaining ones on the right are set with a bit flag indicating emptiness. This is done to respect the minimum necessary power of two size of the tree. In step 6 all buckets are mapped to a leaf and their corresponding AABBs are calculated based on the voxels it contains. In step 7 the tree is generated in a bottom-up fashion by propagating the AABB of each node.

approach to visit its children, instead of the classical formulae used in binary trees where the left child index is calculated as $2n$, and the right child index is calculated as $2n + 1$, being n the parent node index.

We calculate the left and right children as described in Algorithm 2, where $sumOfBase2$ is calculated as a cumulative sum of powers of two up to the current level and $powBase2$ is a power of 2 exponentiation, calculated using a simple and efficient bit shift operation. To improve performance, all $sumOfBase2$ values up to the tree depth can be pre-calculated, instead of calculating them in real-time using a for loop.

If the current node is a leaf node and it is not empty, we get the number of elements contained in this bucket by calculating $offset[i] - offset[i-1]$ where i is calculated as $currentNodeIndex - firstNodeAtDeepestLevelIndex$ and $offset[i]$ is the starting index for its elements on the Morton codes array. For each Morton code within this bucket, we decode it back to its 3D coordinates, calculate its AABB and check for an intersection. If this leaf node is the rightmost child of its parent, then the next node to be visited is its parent index plus one. The ray traversal is shown in Figure 4, and its pseudo-code presented in Algorithm 3.

Algorithm 2: How to calculate the left and right children of a given parent node

```
procedure getLeftChild(node, level):
    int leftmost = sumOfBase2(level) - powBase2(level) + 1
    int rightmost = sumOfBase2(level)
    return node + (rightmost - node) + 2 * (node - leftmost)
      + 1
procedure getRightChild(node, level):
    return getLeftChild(node, level) + 1
```

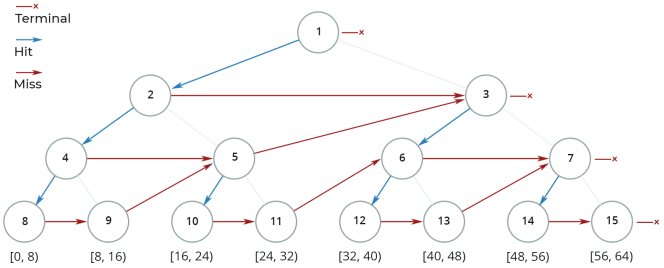


Fig. 4: Tree traversal on a grid 4^3 with bucket size 8.

IV. Results

We performed all tests on a machine equipped with an Intel i7-4770 3.40GHz, 16 GB RAM DDR3, and a GTX 1060 6GB, running on a Windows Seven 64 bits. We performed our tests on Walt Disney Animation Studios Cloud Data Set [19] and OpenVDB Data Sets [20], all of which are publicly available. Namely, Dragon Harvard, Smoke, Explosion, Bunny cloud, and Cloud were chosen. Particularly for the Cloud dataset, we chose two versions: $1/8$ and $1/4$ of its original size. We present the datasets in Figure 1, rendered with our in-house renderer.

We evaluate average tree construction time, average tree traversal time, and memory usage for varying bucket sizes. We compared all evaluations to our implementation of Zellmann et al. [2], named as *Reference*, using a brick size of 8^3. We computed the final construction time for every dataset and bucket size by adding how much time each function took to finish using CUDA events. Traversal time was calculated for each face of the AABB comprising the volume. We cast rays perpendicular to each face pointing towards the volume, as if

Algorithm 3: Bucket LBVH Tree traversal

```
procedure traverse(Ray):
    currentNode = 1
    currentLevel = 0
    while currentNode ≠ numberOfNodes do
        bool miss
        if currentNode is empty then
            miss = true
        else
            miss = intersect(ray, currentNode.AABB)
        end

        if miss is true then
            if currentNode ==
              rightmostNodeOfCurrentLevel then
                break
            end
            if currentNode is parent's rightmost child then
                currentNode = getParent(currentNode) + 1
                currentLevel = currentLevel - 1
            else
                currentNode = currentNode + 1
            end
            continue
        end

        if currentNode is a leaf then
            foreach MortonCode m in this leaf's bucket do
                intersect(ray, getAABB(m))
            end
            if currentNode = numberOfNodes then
                break
            end
            if currentNode is parent's rightmost child then
                currentNode = getParent(currentNode) + 1
                currentLevel = currentLevel - 1
            else
                currentNode = currentNode + 1
            end
        else
            currentNode = getLeftmostChild(currentNode)
            currentLevel = currentLevel + 1
        end
    end
```

seeing the volume perfectly aligned with the screen, and each voxel occupying a single pixel. We measured the time taken to complete each tree traversal for each ray and then summed all of them to obtain an average traversal time for all rays. Once we finish all faces, we averaged the averages of all faces.

We calculated memory consumption using the number of nodes on each tree multiplied by its size in bytes. Because in our case, we store Morton codes and offsets in an auxiliary array, we also added these. Figure 5 shows the memory usage of our novel algorithm for the three largest grids we tested as well as for our Zellman et al. implementation. Note that the smaller a bucket is, the higher our memory overhead is. This happens because we need to store an offset for each bucket and, with more buckets, more offsets to store. The only constant part for each bucket size is the Morton codes array, reflecting the grid size in terms of non-empty voxels.

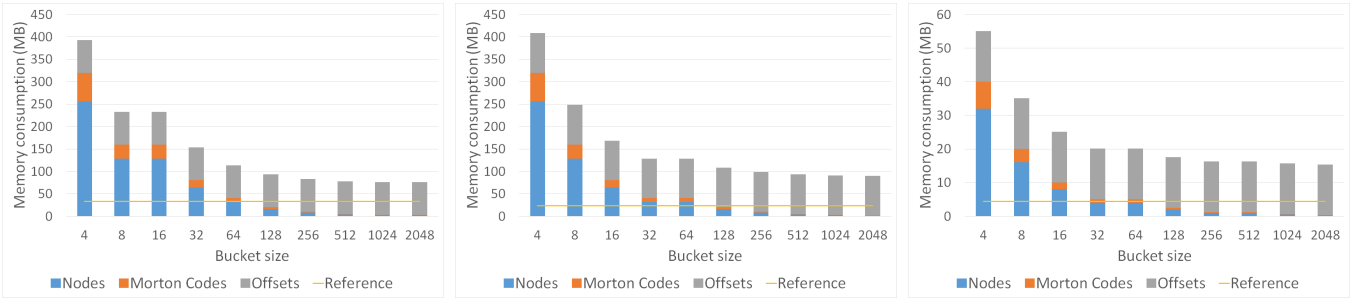
Our memory usage has a higher overhead compared to [2] that comes from storing Morton codes for each non-empty voxel. In contrast, they store a Morton code for each brick of size 8^3, thus saving more memory than our implementation. However, since GPUs nowadays have increasingly memory sizes, and for many real-time applications such as games, the grid sizes are smaller, it does not impact as much. For example, this is shown in an implementation done by Andrew Schneider [21], where his real-time volumetric cloudscapes used 3D textures sizes of 32^3 and 128^3 in the game Horizon Zero Dawn. His technique had limited grid sizes due to rendering performance when raymarching and storage limitations due to other components of the game. Many games and game engines that followed his technique also had limitations in grid sizes to meet real-time constraints, such as Frostbite [22] and Red Dead Redemption 2 [23]. The trade-off between performance and memory may be well suited for such applications where grid sizes are smaller or memory is not a limitation.

Following on these limitations, we can project a case scenario with our Bucket LBVH algorithm. Given a grid of resolution 128^3 with all voxels having a density higher than zero comprised of a 32 bits float and a bucket size of 32, we can calculate the total memory overhead from our algorithm: 8.38 MB for all 2,097,152 Morton codes, 1.04 MB for all 131,071 nodes and 0.26 MB for all 65,536 offsets, totaling 9,68 MB for the whole structure. Therefore, the total memory fits in any GPU nowadays, as shown in the Steam hardware survey [24], where less than 12.87% of its user base have GPUs equipped with less than 2GB of VRAM, which implies more than enough space to store these grids while leaving plenty of room to all other data a game may need.

Furthermore, even though Zellman et al. works on top of bricks, which in consequence, consume less memory, they have to loop through all voxels to create the struct that encapsulates each brick. When sorting all these Morton codes, we compared looping through an array of integers versus looping through an array of structs. The former performed better than the later when using thrust to sort, which partially contributes to our construction time gains.

Table I shows the best and worst construction times for each bucket size alongside its performance gains. We improved construction performance by a varying degree of 3x-12x times more. On average, our construction time improved performance by 9.3 times. As expected, our worst cases were on smaller bucket sizes, since the smaller a bucket is, the deeper a tree is and longer it takes to generate it.

Table II showcases the best average traversal time for each grid and bucket size alongside its performance gain. As a rule, the smaller a bucket, the better balanced the tree is, and thus tree traversal takes less time. With bigger buckets, which contains more voxels, we must perform more intersection tests, which is costly performance-wise. The caveat here is, the smaller a bucket, the worse construction time is. Not only is construction time worse, but also memory consumption. For our best cases in traversal times, with bucket sizes of 4, we performed 2.2 times better on average. This becomes clear in Figure 6, where, from a given point, we note that increasing bucket sizes makes our algorithm perform worse in traversal times than Zellman et al. This performance is a trade-



(a) Case I - Bunny (b) Case II - Cloud quarter (c) Case III - Explosion

Fig. 5: Memory usage in MB for each bucket size for three of the largest grids. Reference is fixed at a brick size of 8^3.

off between traversing the tree, which is $O(log(n))$, traversing the array of voxels within each bucket, which is $O(n)$, and AABB intersection tests.

Our best-case scenario lies around bucket sizes of 64 and 128, where we have better traversal times, winning in most cases against our reference, while also getting excellent construction times and small memory overhead. An important point here is that we only use power of two bucket sizes. This is due to the nature of Morton codes and its Z shape in 3D space, where power of two sizes helps preserve data locality better when grouping them in buckets.

V. Conclusion and Future Work

We have presented a novel Bucket LBVH algorithm for constructing and traversing sparse volumes. Our algorithm improves construction performance by an average of 9.3 times on tested grids. Some of its limitations are high memory overhead with smaller buckets and worse traversal performance for too large bucket sizes.

For future work, we aim to test larger grids, with resolutions of 1024^3 and onwards. Another point of improvement is to experiment with trees of other dimensions such as quadtrees and octrees, to see how well they perform for each bucket size. With these other tree dimensions, we could also correlate and test a heuristic to determine which tree model is best for each volume based on how sparse it is.

A next step to further verify the robustness of our algorithm would be to test grids from medical data and animated fluids. Another significant factor to play a role in future work is to evaluate performance and implementation on shaders and measure not only traversal times but render times using volumetric equations.

Furthermore, our implementation is publicly available at github.com/ibfernandes/bucketlbvh.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. We also thanks CNPq for scholarship funding for the first author.

References

[1] A. Schneider, "Real-time volumetric cloudscapes," in *GPU Pro 7*, W. Engel, Ed. CRC Press, 2016, pp. 97–127.

[2] S. Zellmannn, M. Hellmann, and U. Lang, "A linear time bvh construction algorithm for sparse volumes," in *2019 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 2019, pp. 222–226.

[3] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, p. 343–349, Jun. 1980.

[4] S. M. Rubin and T. Whitted, "A 3-dimensional representation for fast rendering of complex scenes," in *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, 1980, pp. 110–116.

[5] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM siggraph computer graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[6] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Transactions on Graphics (TOG)*, vol. 26, no. 1, pp. 6–es, 2007.

[7] R. Torres, P. J. Martín, and A. Gavilanes, "Ray casting using a roped bvh with cuda," in *Proceedings of the 25th Spring Conference on Computer Graphics*, ser. SCCG '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 95–102. [Online]. Available: https://doi.org/10.1145/1980462.1980483

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

[9] J. Pantaleoni and D. Luebke, "Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry," in *Proceedings of the Conference on High Performance Graphics*, 2010, pp. 87–95.

[10] G. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966. [Online]. Available: https://books.google.com.br/books?id=9FFdHAAACAAJ

[11] F. A. Sergio Murguia, Arturo Garcia and L. Reyes, "Stack-less lbvh traversal for real-time ray tracing," in *Proceedings of CGI 2011*, 2011.

[12] L. R. Sergio Murguia, Francisco Avila and A. Garcıa, "Bit-trail traversal for stackless lbvh on directcompute," in *GPU Pro 4: Advanced Rendering Techniques*, W. Engel, C. Oat, C. Dachsbacher, M. Valient, W. Bahnassi, and S. St-Laurent, Eds. AK Peters, 2013, pp. 319–336.

[13] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, "Efficient stack-less bvh traversal for ray tracing," in *Proceedings of the 27th Spring Conference on Computer Graphics*, ser. SCCG '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 7–12. [Online]. Available: https://doi.org/10.1145/2461217.2461219

[14] T. Fogal, A. Schiewe, and J. Krüger, "An analysis of scalable gpu-based ray-guided volume rendering," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 43–51.

[15] J. Beyer, M. Hadwiger, and H. Pfister, "A survey of gpu-based large-scale volume visualization," in *Eurographics Conference on Visualization (EuroVis)(2014)*. IEEE Visualization and Graphics Technical Committee (IEEE VGTC), 2014.

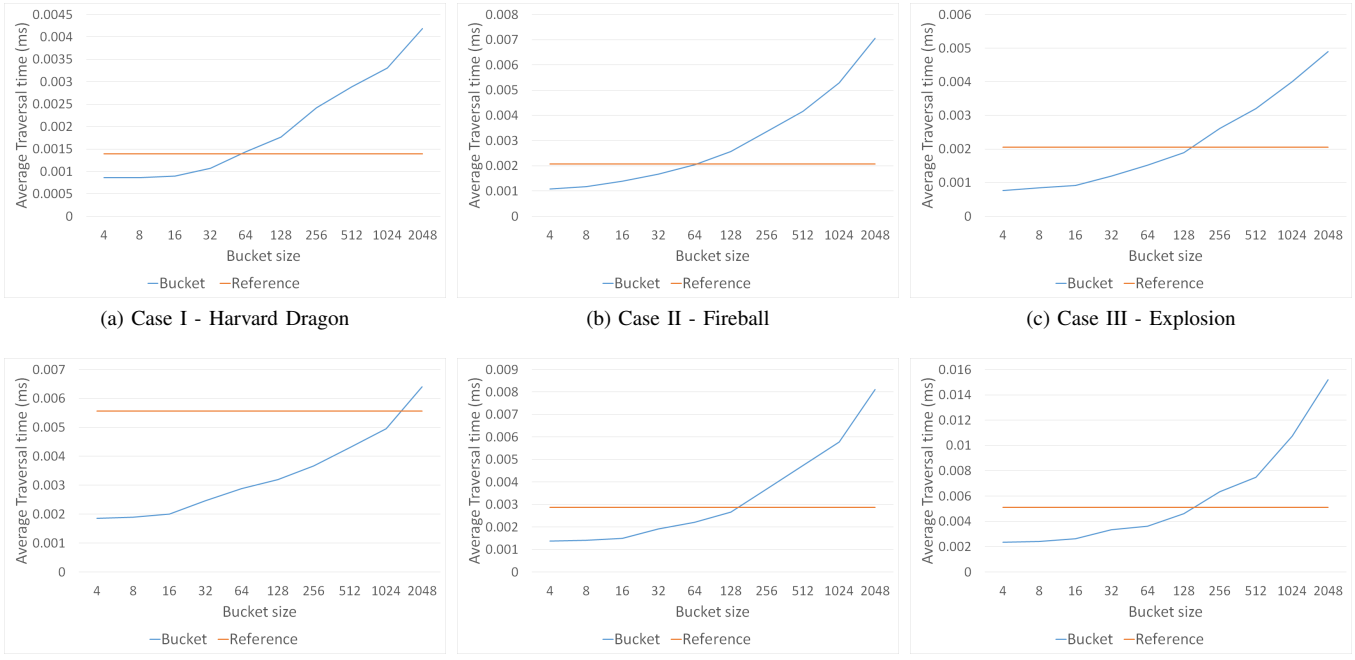
[16] D. Stroter, J. S. Mueller-Roemer, A. Stork, and D. W. Fellner, "Olbvh: octree linear bounding volume hierarchy for volumetric meshes," *The Visual Computer*, 2020.

TABLE I: Best and worst cases in construction time (ms)

Grid	Resolution	Worst case Bucket size	Bucket LBVH	Best case Bucket size	Bucket LBVH	Reference	Performance Gain
Havard Dragon	101x50x100	4	1.31603	2048	1.34623	4.82716	3.6x
Cloud Eighth	249x169x306	4	18.4432	2048	13.3931	114.387	6.2x – 8.5x
Fireball	268x298x267	4	25.1156	2048	20.5236	198.679	7.9x – 9.6x
Explosion	328x349x311	4	35.4059	2048	30.6663	333.416	9.4x – 10.8x
Cloud Quarter	497x337x612	4	130.448	2048	89.9799	1015.84	7.78x – 11.2x
Bunny Cloud	576x571x437	4	141.325	2048	119.868	1468.04	10.3x – 12.2x

TABLE II: Best cases in traversal time (ms)

Grid	Resolution	Bucket size	Bucket LBVH	Reference	Performance gain
Havard Dragon	101x50x100	4	0.000857581	0.00138955	1.62x
Cloud Eighth	249x169x306	4	0.00137175	0.00286178	2.08x
Fireball	268x298x267	4	0.0010806	0.00207262	1.91x
Explosion	328x349x311	4	0.000760094	0.00204735	2.7x
Cloud Quarter	497x337x612	4	0.00236053	0.00510869	2.16x
Bunny Cloud	576x571x437	4	0.00185723	0.00555913	3.0x



(a) Case I - Harvard Dragon (b) Case II - Fireball (c) Case III - Explosion

(d) Case IV - Bunny cloud (e) Case V - Cloud eighth size (f) Case VI - Cloud quarter size

Fig. 6: Average traversal times in milliseconds for each grid and bucket size configuration. The yellow line represents Zellman et al. traversal time, with brick size fixed at 8^3, and the blue line is the Bucket LBVH traversal time.

[17] N. Bell and J. Hoberock, "Chapter 26 - thrust: A productivity-oriented library for cuda," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 359 – 371. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123859631000265

[18] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, C. Dachsbacher, J. Munkberg, and J. Pantaleoni, Eds. The Eurographics Association, 2012.

[19] Walt disney animation studios cloud data set. [Online]. Available: https://www.technology.disneyanimation.com/clouds

[20] Openvdb data sets. [Online]. Available: https://www.openvdb.org/download/

[21] W. Engel, *GPU Pro 7 - Advanced Rendering Techniques*, 1st ed. A K Peters/CRC Press, 2016.

[22] S. Hillaire. (2016) Physically based sky, atmosphere and cloud rendering in frostbite. [Online]. Available: https://media.contentapi.ea.com/content/dam/eacom/frostbite/files/s2016-pbs-frostbite-sky-clouds-new.pdf

[23] F. Bauer. (2019) Creating the atmospheric world of red dead redemption 2: A complete and integrated solution. [Online]. Available: http://advances.realtimerendering.com/s2019/index.htm

[24] (2020) Steam hardware and software survey. [Online]. Available: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam