DOI: 10.1111/cgf.14758
EUROGRAPHICS 2023 / K. Myszkowski and M. Nießner
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 2

Parallel Transformation of Bounding Volume Hierarchies into Oriented Bounding Box Trees

N. Vitsas[1], I. Evangelou[1], G. Papaioannou[1], A. Gkaravelis[1]

[1]Department of Informatics, Athens University of Economics and Business, Greece

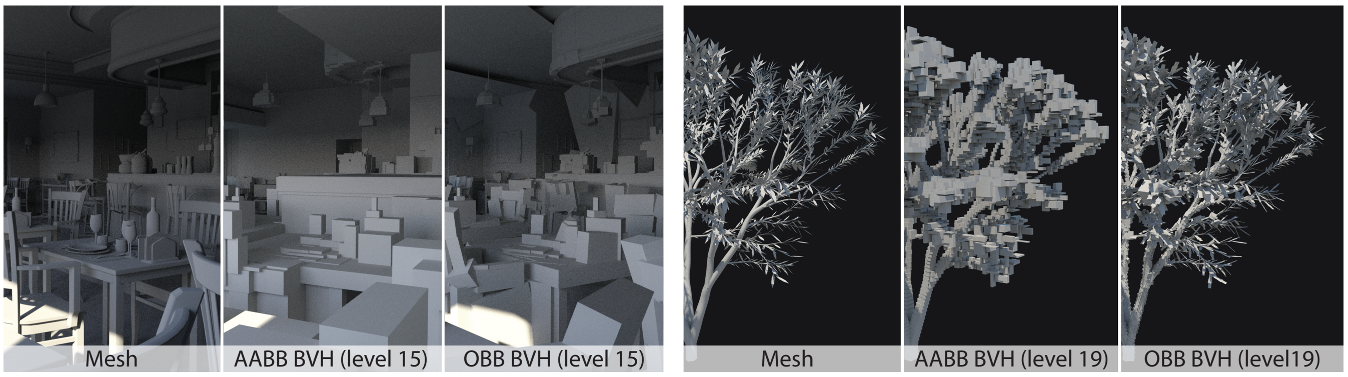


Figure 1: *Examples of direct transformation of an AABB BVH into the corresponding OBB hierarchy. The improved fitting potential of OBBs can help minimise internal and leaf node intersections during tree traversal for ray tracing.*

Abstract

Oriented bounding box (OBB) hierarchies can be used instead of hierarchies based on axis-aligned bounding boxes (AABB), providing tighter fitting to the underlying geometric structures and resulting in improved interference tests, such as ray-geometry intersections. In this paper, we present a method for the fast, parallel transformation of an existing bounding volume hierarchy (BVH), based on AABBs, into a hierarchy based on oriented bounding boxes. To this end, we parallelise a high-quality OBB extraction algorithm from the literature to operate as a standalone OBB estimator and further extend it to efficiently build an OBB hierarchy in a bottom up manner. This agglomerative approach allows for fast parallel execution and the formation of arbitrary, high-quality OBBs in bounding volume hierarchies. The method is fully implemented on the GPU and extensively evaluated with ray intersections.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Ray tracing;*** *Visibility; Mesh geometry models;*

1. Introduction

Applying queries on geometric primitives of arbitrary type and complexity is a common task that emerges in numerous research and industry fields. One such dominant field is ray tracing, which is at the core of most high-quality algorithms for image synthesis and light simulation in general. Object-based partitioning of a scene with bounding volume hierarchies (BVH) has been the dominant strategy for fast ray tracing in recent years. A bounding volume can be chosen to be any closed geometric shape, such as an axis-aligned bounding box (AABB) [SFD09], an oriented bounding box (OBB) [CWK10, TY09], a capsule, a sphere, a k-DOP [KHM*98] and others, with many of them finding great success in specialised applications. In this particular work, we focus on the OBB-based representation of tree nodes. As a bounding volume, OBBs have a higher tight-fitting potential than AABBs and are moderately more complex to handle. They are also more robust to orientation changes and would greatly benefit scenarios with rigid-body animations, since only the transformation of the box needs to be updated, thus avoiding the need for a complete re-computation of a node's OBB. They are also very fast to refit in the case of mesh deformations.

Typical state-of-the-art hierarchies use axis-aligned bounding boxes due to their simplicity, generality and efficiency in terms of bound estimation and intersection. Nevertheless, orientation-aware bounding volumes have been extensively used to accelerate ray-tracing and other interference computations [RW80, GLM96, LAM01], such as collision detection and boolean operations, as well as view frustum culling [AM00]. The two main concerns regarding the construction of an OBB hierarchy are a) the fast and robust calculation of a tightest-fitting OBB and b) the efficiency of ray-OBB intersection and hierarchy traversal. In this paper, we target the problem of fast OBB construction, both as a single bounding volume and a hierarchical representation. We implement a fast parallel OBB estimator and propose an agglomerative method for the conversion of AABB hierarchies to OBB-based ones.

Despite the advantages of using an AABB tree, it has been shown that under certain scenarios these can dramatically overestimate the enclosing shape of a cluster. This is especially apparent when it comes to elongated and arbitrarily oriented clusters of primitives (e.g. branches, leaves, hair strands). Also, in the case of rigid animation, OBB hierarchies do not require any refitting to the animated data, making them vastly superior to AABBs. While OBB-based representations are considered a theoretically superior choice, resulting in tighter bounds of their enclosing geometry and therefore fewer false positive intersections, the extraction of such hierarchies is generally computationally expensive. Parallelisation and scaling of an OBB builder is not trivial out of the box, and this is one of the two major contributions of our work. Furthermore, the hierarchy traversal requires that the query entity (e.g. a ray or bound) be transformed to the local reference frame of the node's OBB prior to intersection, an operation whose cost often outweighs the benefits. This has led to ray-OBB intersections not being supported by today's ray tracing frameworks at the node level, but only at the boundary between the top and bottom parts of a two-level hierarchy. We show next that with careful handling of transformations and memory within the OBB traversal kernel, the traversal overhead can be minimised.

The proposed method is executed as a post-process step over an existing bounding volume hierarchy. In that sense, the algorithm used for generating the initial BVH is orthogonal, ensuring compatibility with any existing fast GPU and CPU builder. In this work we focus on binary hierarchies, which have some of the most common and robust builders, although the idea is equally applicable to wider hierarchies. We contribute to the current state of the art by a) presenting a fast parallel implementation for the extraction of OBBs from unordered point sets and b) proposing a parallel agglomerative algorithm for the computation of high-quality OBBs for all of the nodes of an existing BVH.

The remainder of the paper is structured as follows. In Section 2 we briefly cover the literature of OBBs and its uses in ray tracing and beyond as well as the state-of-the-art in optimising an existing BVH. In Section 3 we cover the parallel implementation of both the standalone OBB extraction from point sets and the agglomerative approach for transforming existing hierarchies. In Section 4, we evaluate the performance characteristics of the resulting structures on various state of the art builders. We conclude with Section 5, where we summarise our contribution and discuss limitations and future work.

2. Related Work

Oriented bounding box estimation. Here, we revisit the idea of the OBB calculation from an input point set, for which we first briefly discuss notable existing approaches to this generic problem. The initially proposed algorithm, which calculates an optimal tightest fit box, albeit the slowest, was demonstrated by O'Rourke [O'R85]. A popular approach based on the intrinsic relationships of the input samples is based on a purely mathematical formulation that uses Principal Component Analysis (PCA) [AW10] while an ε-approximation algorithm specifically designed for $\mathbb{R}^3$ was designed by [BHP01]. More recently, Chang et al. [CGM11] approached the problem as an unconstrained optimisation problem, computing the orientation of an optimal OBB by a hybrid global optimisation algorithm that searches in the space of rotation matrices.

For this work, we shift our focus to another approximate approach abbreviated to DiTO [LK11], which has demonstrated a nice balance between bounding volume quality and run-time complexity for triangular meshes, compared to PCA and other approximate methods. We modify the algorithm and map it to any parallel computing device, such as a GPU architecture, and most importantly, adapt it for the complete construction of bounding volume hierarchies of OBB-based tree nodes.

Bounding volume hierarchies. High performance BVH construction has been extensively studied and surveyed [MOB*21] in the past years. In this field of research and mostly targeting GPU acceleration, the most common approach is to construct a linear bounding volume hierarchy [LGS*09,Kar12,Ape14]. Based on this method many algorithms have been proposed over the years that are constantly pushing the state-of-the-art in terms of build time and resulting tree quality. Gu et al. [GHFB13] presented one of the earliest approximate agglomerative techniques for fast BVH construction. Karras et al. [KA13] demonstrated an agglomerative BVH optimisation process based on restructuring small collections of nodes called treelets, which Domingues et al. [DP15] improved on in terms of construction time, while at the same time, widening the treelet size based on a greedy clustering metric. In a similar manner, Meister and Bittner [MB18b], proposed a bottom-up clustering algorithm for BVH construction based on spatially local neighbours and a parallel topology reorganisation algorithm [MB18a]. Our method contributes in the category of optimisation techniques of already constructed hierarchies and is orthogonal to other approaches.

OBB hierarchies. Exploiting OBB-based nodes for a bounding volume hierarchy is not an entirely new concept in ray tracing. Closely related to our work, there has been research towards hair and fur rendering by Woop et al. [WBW*14] where objects predominantly consist of thin, long primitives and are well suited for this type of bounding volume intersection queries. This was achieved by constructing a hybrid AABB/OBB BVH in a top-down fashion for a CPU ray tracer in a production environment.

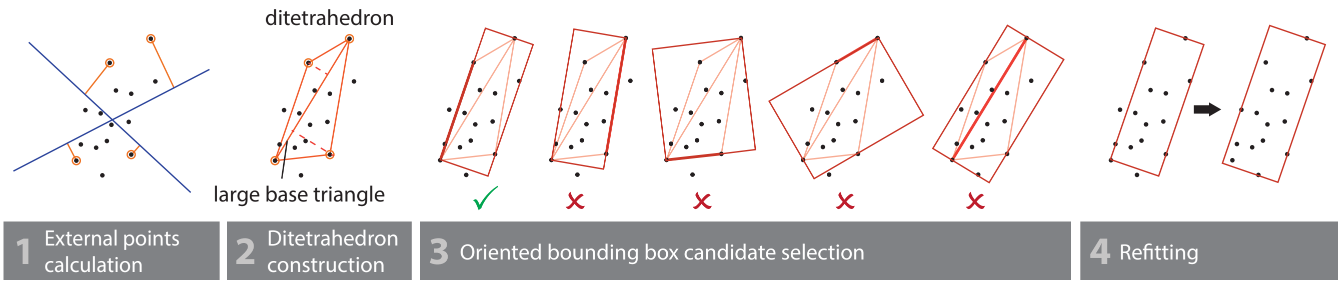

ditetrahedron
large base triangle
✓
✗
✗
✗
✗
1 External points calculation
2 Ditetrahedron construction
3 Oriented bounding box candidate selection
4 Refitting

Figure 2: *The principle of operation of the DiTO oriented bounding box construction method, illustrated in two dimensions.*

Our approach does not limit itself to these model inputs and can be implemented on top of arbitrary primitives defined by points. Orientation-aware transformations at the primitive level for thin, oblong geometry, were also exploited by Wald et al. [WMZ*20] in order to take advantage of the existing hardware ray tracing cores and instancing capabilities of modern APIs.

3. Method Overview

Finding a tightly fitting oriented bounding box for a set of geometric primitives requires the calculation of an orientation, a midpoint and the box's extents, so that any primitive in the bounded set is fully contained withing the OBB. Although an exact, polynomial time algorithm for OBB computation exists [O'R85], it is not easily parallelisable. Approximate methods, such as those based on PCA can be used, instead. However, it has been shown both empirically, by Larson et al. [LK11], and through in-depth analysis, by Dimitrov et al. [DKKR09], that there is an upper bound on the accuracy of PCA-based OBB extraction in $\mathbb{R}^2$ and $\mathbb{R}^3$. In the particular case of BVH construction for 3D meshes, small inaccuracies, propagated up the hierarchy, can be very problematic. For this reason, we chose DiTO [LK11], a method specifically designed for geometry meshes.

We start by parallelising the DiTO algorithm to run on the GPU, in order to compute a single OBB from an unordered point set. This can be used as a standalone OBB builder for arbitrarily large datasets, but also for instance pre-alignment in two-level hierarchies. The core concept is then adapted to agglomeratively operate on the nodes of a hierarchy and output an OBB hierarchy.

3.1. The Ditetrahedron OBB algorithm (DiTO)

DiTO was designed to be applicable to point clouds, polygon meshes, or polygon soups, without any need for an initial convex hull generation. The algorithm is based on processing a small, constant number of characteristic vertices selected from the input model. The selected points are then used to construct a representative simple shape, a double tetrahedron, or *ditetrahedron*. A suitable orientation for a tight-fitting bounding box can be efficiently derived from the edges of this shape. To quickly explain the algorithm, let's consider a point set $\mathcal{P}$ of N points in 3D space for which we want to find a tightly fitting OBB. The DiTO algorithm can be broken down into 4 distinct steps, illustrated also in Figure 2, as follows.

External Points Calculation. The algorithm begins by finding the vertices in $\mathcal{P}$, which have the maximum or minimum projection on a set of K standard axes. There is no restriction as to which set of axes one should use, but the authors suggest choosing a number K that allows to uniformly sample the domain of orientations and potentially accelerate subsequent calculations. Based on the author's recommendation, we implement the DiTO-14 variant, which uses $K = 7$ standard projection axes. This results in a collection of K min/max value pairs and a multiset $\mathcal{S}$ of $2K$ external points, in total. When a point corresponds to either a minimum or a maximum projection on more than one axes, it is inserted in $\mathcal{S}$ multiple times.

Ditetrahedron construction. The next step of the algorithm operates solely on the subset $\mathcal{S}$ of vertices. A *large base* triangle is formed from 3 of the candidate points in $\mathcal{S}$, using the farthest vertices as one edge of the triangle and the farthest point to that edge as the third triangle vertex. From this triangle, the algorithm proceeds to create two joined tetrahedra (a ditetrahedron) using as apexes the opposite farthest points from the triangle plane.

OBB candidate selection. The DiTO algorithm iterates over all seven facets of the ditetrahedron, examines all triangle-aligned local reference frames as the axes of a candidate tightly fitting OBB, including an axis-aligned box, and chooses the one that results in the smallest surface area. Very often, the ditrahedron is degenerate, for instance when the points in $\mathcal{S}$ are fewer than the six vertices of the ditetrahedron. However, this causes no problems, since we only use the individual triangular sides and not the polyhedron as a whole. Collapsed triangles are simply discarded.

OBB refitting. The previous step has defined the orientation of a good OBB, which must be now refitted to include all points in $\mathcal{P}$, since the face orientation of the OBB is derived from the ditetrahedron's facets rather than the min/max axial projections of the current local reference frame. The OBB extents are calculated by iteratively determining the minimum and maximum projections of all points in $\mathcal{P}$ on the local OBB axes.

3.2. Parallel OBB Extraction

The standard DiTO algorithm can be parallelised on the GPU. The first and last steps of the DiTO algorithm essentially comprise a reduction operation to determine minimum and maximum projections on a set of axes: K for the first step, 3 for the final one. Such a reduction is implemented very efficiently on the GPU, using shared memory or warp-level shuffle operations. The resulting block-level results are written to global memory with fast atomic operations.

The intermediate step is significantly less involved, with purely deterministic iterations over the faces of the ditetrahedron to locate the best candidate OBB. This is a very fast step, which can be efficiently performed in both CPU and GPU. To keep memory resident on the GPU between the expensive steps, we simply choose to perform the operations on the GPU with a small kernel launch.

The overall, parallel algorithm is extremely fast, even for very large point sets (see Table 1 in Section 4). Due to the fast OBB calculation time, apart from the OBB extraction for arbitrarily large meshes, it can be used as a potential pre-alignment step during the bottom-level construction of a two-level hierarchy, or in geometric tasks such as object classification, point cloud registration, etc.

3.3. OBB Extraction for Bounding Volume Hierarchy Nodes

We wanted our agglomerative approach to be applicable to any existing BVH hierarchy without imposing restrictions to the underlying ordering of the primitives within the leaves. A specific ordering, which would be exploitable in our case, is unfortunately not guaranteed in many of the fast state-of-the-art builders. The proposed algorithm works in parallel, starting from the leaves of the given hierarchy and modifies the node bounds, leaving the existing node linkage unchanged. In order for such an agglomerative method to operate efficiently, an important property is to maintain a constant storage payload during the hierarchy traversal and keep the respective computations complexity constant.

There are two easily attainable requirements for the method. First, each node must refer to its parent, in order for the bottom-up traversal to be feasible. This information is very common in builders [DP15, LGS*09] and is easy to calculate if its not readily available [SFD09]. Second, leaves need to have access to vertex data of the stored primitives. This information is readily available for meshes or easy to calculate for many non-triangulated primitives. Alternatively, already computed bounds, or representative points can be provided as leaf data, similar to the case of AABB trees.

The key insight behind an agglomerative DiTO version is that each tree node only needs to gather and update the minimum and maximum projection vertices for the K standard axes, as these propagate up the hierarchy. An initial set of K minimum and maximum projection coordinates are determined at the leaves in parallel, which are subsequently iteratively routed and merged upwards, in isolation. Projection coordinates are stored in global memory and are unique for each node. An illustration of the parallel bottom up traversal can be seen in Figure 3.

After having calculated the required projections and vertices in the parallel bottom-up pass, the OBB calculation according to the second and third step of the DiTO method for each node is concurrently and independently performed. On the GPU, these are implemented as single kernel launch. At the end of this step, the three OBB axes and their extents for each node are stored in global memory.

The next step is the refitting operation for each node in order to ensure the full enclosing of their children. Normally, each node requires an iteration over all primitives in its stemming sub-trees, to adjust the OBB extents. However, such a pull operation is extremely inefficient in any parallel paradigm. Instead, we reverse the process and make it primitive-driven. Each vertex from the leaf nodes iteratively traverses the path to the root independently in a data parallel manner and updates the encountered node extents, as necessary using atomic min/max operations. We are careful to initialise the minimum and maximum values from the projections of the previously calculated OBB to avoid unnecessary atomic updates. The refitting process is fast and embarrassingly parallel, with full core utilisation up to the root. Its speed and its very small memory footprint, indicate that node data can be made resident and reused if the object is dynamically morphed.

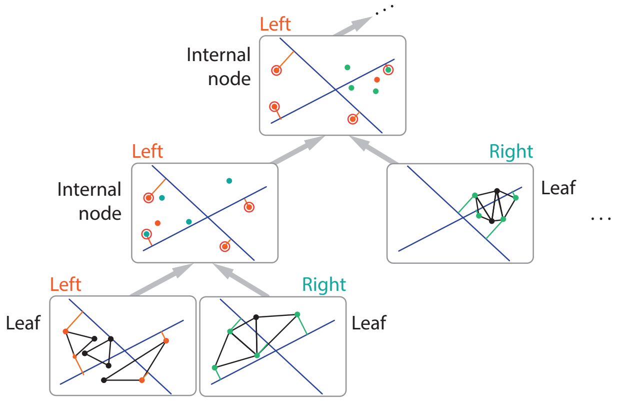


Figure 3: *Parallel OBB hierarchy construction - concurrent propagation of projection extrema.*

A final step determines and stores data for the traversal of the tree, i.e. the root-to-node transformation matrix, so that intersections can be computed accurately and fast, with a unit AABB, instead of arbitrary OBBs. The resulting matrix is a typical scaling and change of reference frame transformation. In our GPU implementation, this step is performed as a separate kernel launch.

Finally, it is worth mentioning that the OBB hierarchy conversion process is directly applicable to wider trees, not just binary hierarchies.

3.4. Implementation Details

GPU implementation. We implemented the four-stage process of DiTO (see Figure 2) with four highly parallel kernels. The first kernel is launched over all leaves of the hierarchy. We calculate the projection values and store them in global memory and move on to their parents. The last child to reach the parent during parallel execution fetches the projection values for all siblings (two here) from global memory and proceeds to further resolve the min and max values to be propagated to its ancestor. This strategy is executed until a single thread reaches the root.

In the second kernel launch, each node uses the projections and corresponding vertex indices stored in global memory from the previous pass. A number of instances of the DiTO algorithm equal to the number of tree nodes, are executed in parallel, implementing steps 2 and 3 of Figure 2.

The third kernel performs the refitting of the resulting OBB for each node. We again launch a kernel over all leaf nodes and ascend the hierarchy. In contrast to previous bottom-up passes, no node exits upon reaching its parent. This allows to robustly and accurately grow the OBB bounds taking into account all enclosed primitives of an internal node, at any level. Therefore, every single thread persists until reaching the root. To avoid excessive stalls by constant atomic updates to the bounds of each OBB, we make sure to perform the updates only if a fast point-in-OBB check fails. This is a highly data-parallel step that fully employs the GPU. It is also the fastest of the 3 kernels, which is very important as it is the only step that is required at runtime, in case of mesh deformation.

Finally, a separate kernel is launched independently over all nodes, which calculates the transformation matrix required for the root-to-node transformation, using the approach presented in [SAVBCNRM21].

An important mechanism that we leverage throughout various parts of our implementation is a fast bottom-up traversal of the hierarchy using parent connections. We use the approach taken by Karras [Kar12] that synchronises accesses to the parent node by its children using a per-node atomic flag in global memory. Where needed, the child that reaches the parent node first, marks the node as *visited* and this ensures that only a single child operates on each node. This mechanism can further be exploited in order to store more meaningful information. We use it to store the child that did not enter the node in order to quickly get access to the sibling.

Deferring the external point identification. For the standalone OBB calculation process, the external points calculation step of the original DiTO algorithm, keeps track of the external points along with the update of the minimum and maximum axial projections. In a GPU environment, this is not efficient to implement. We address this by observing that the actual vertex corresponding to a min/max projection is only required *after* the sorting. Therefore we defer the identification of external points to a separate kernel launch, after the min/max axial projection values are known. We launch the kernel over the entire point set and atomically update the external point index, when a point has exactly the same projection as the calculated min and max.

Degenerate OBBs. In our method, we allow for the calculation of OBBs, at any level and with no restrictions on the number or type of primitives in the case of leaf nodes. Therefore, we need to handle special cases, such as single-primitive leaves, where OBBs collapse (e.g. triangles), resulting in non-invertible transformations. Such cases are easy to identify by their zero-length extents in at least one of the OBB axes. We simply inflate the collapsed sides of the OBB by a fraction of the smallest non-collapsed side.

Hierarchy collapse. Most GPU BVH optimisers perform an additional step in which terminal subtrees are collapsed into multi-primitive leaves, if the SAH cost becomes lower. Meister et al. [MB18a] include a fairly thorough description of the implementation of such a routine in a bottom-up manner. This step is also orthogonal to our optimisation pass. A key observation for this step is that it does not perform any reorganization of the hierarchy and therefore does not affect the bounds of any remaining node. This effectively means that no OBBs need to be recalculated, allowing the collapse to occur either before or after the OBB BVH transformation. Please keep in mind that current collapse implementations create primitive clusters solely based on spatial proximity. Though very helpful for AABB hierarchies, large collapses may degrade OBB tree traversal performance, by increasing the volume and overlap of node OBBs, as well as flattening clusters of thin, long primitives.

3.5. Tree traversal

Our goal was to introduce as few modifications to the hierarchy traversal method as possible so that it operates on OBBs instead of AABBs. We use the persistent *"while-while"* kernels presented by Aila et al. [AL09] as a baseline. These require that the intersection information for the children are stored directly in the parent for efficient access. For AABBs these data correspond to the min and max points of the child's bounds. For OBBs, the inverse transformation for each child is needed which we store as a 4x3 matrix. We use a struct-of-arrays (SoA) internal node memory layout for more efficient access. For leaf nodes, only the corresponding range of the primitive data is required. At the code level, we only need to add routines for the transformation of each ray's origin and direction. Finally, to speed up ray-OBB intersections we also use a custom unit-AABB intersection routine that directly intersects with the $[-0.5,0.5]^3$ cube, following the ray transformation with the matrix derived by Sabino et al. [SAVBCNRM21].

4. Evaluation

We evaluate our method using two distinct sets of experiments. The first set is a performance evaluation of the standalone GPU implementation of DiTO that operates directly over a flat set of input points and extracts a high-quality oriented bounding box. The second set of experiments is used for the evaluation of the conversion of an AABB hierarchy into an OBB one, specifically targeting ray tracing performance. Throughout all experiments, we treat the input as raw triangles, flattening any transformed instances of sub-parts. From that, we either build a single OBB (first set of experiments) or an AABB hierarchy, which we then convert to an OBB tree. We use separate, optimised ray tracing kernels for the AABB and OBB hierarchy traversal, respectively. As a test set, we use a collection of 10 scenes. 6 of these are single object setups, typically found in the bottom level of a two-level hierarchy. The other 4 represent flat, non-hierarchical large environments.

All timings were recorded on an NVIDIA RTX 3080Ti with 12GB of VRAM and an Intel i7 12700K CPU with 32GB of RAM. We use NVIDIA CUDA version 11.7 for the GPU implementation of the method and the GPU hierarchy traversal kernels.

4.1. Parallel standalone DiTO

Timings for the standalone execution of DiTO on the 6 single-object cases can be seen in Table 1. The results show a significant performance improvement over the original CPU implementation, favouring large inputs, while maintaining linear scaling. This indicates that there is an interesting potential in applying this fast

OBB calculation as an automatic pre-alignment step for the bounds of bottom-level acceleration structures in two-level BVH hierarchies. On average, the reference CPU implementation consumes 151K points/ms compared to the 2.4M points/ms of our GPU implementation, which constitutes an average 15.7× speedup.

Scene	Point Count	Timings (ms)	
		Reference	Ours
Trees	270K	1.88	0.28 (6.71×)
Abstract	370K	2.38	0.3 (7.93×)
Dragon	420K	2.68	0.33 (8.12×)
Hairball	1.5M	9.21	0.55 (16.74×)
Crown	1.8M	11.66	0.67 (17.19×)
Sheep	23.3M	149.23	5.65 (26.41×)

Table 1: *Extraction of a single OBB on the GPU using a parallel implementation of the DiTO algorithm. Timings on the CPU were captured using the reference implementation of the DiTO algorithm as provided by the authors at* http://www.gameenginegems.net/geg2.php.

4.2. BVH Conversion for Ray Tracing

Here we investigate the impact on build time and ray tracing performance for the AABB-to-OBB hierarchy conversion, used as a post-processing step to construct a complete OBB tree from the hierarchical primitive clustering produced by three high-performance builders. More specifically, we apply our OBB transformation kernels to a) the LBVH builder, as presented in [Kar12] that has fast build times but lacks in ray traversal speed and is typically used for the top-level part in two-level hierarchies, b) ATRBVH that trades build time for higher overall ray tracing performance and c) SweepSAH-BVH [MB90] that has the slowest build performance, due to its top-down exhaustive search, but the highest ray traversal performance.

For the ATRBVH builder, we use the publicly available implementation with the default hyperparameters suggested by the authors and for the case of SweepSAH-BVH we use the public implementation from [SFD09]. For the latter, the build process takes place on the CPU and the resulting hierarchy is transferred to the GPU for further consumption by our framework. For both ATRBVH and SweepSAH-BVH, the Surface Area Heuristic [MB90] is optimised using a traversal cost of 1.2 and an intersection cost of 1. Finally, all builders use the collapse operation we discussed in Subsection 3.4, based on the same SAH metric to create leaves of up to 8 primitives.

Our experimental setup consists of the 10 scenes flattened as single meshes, as shown in Figure 4. For the 6 of them that represent single objects, we trace paths from viewpoints outside their bounds. For the rest (*Powerplant*, *Bistro Interior*, *Bistro Exterior* and *Temple*), primary rays start from within the respective modelled environment. The size of the scenes ranges from 170K to 12.7M triangles.

The results of our tests are summarised in Figure 4. We first present the BVH conversion time in milliseconds for every test scene. In the second block of the figure, we measure ray intersection times on the GPU for the collapsed AABB-based output hierarchy of the primary builder and the corresponding times for the hierarchy as transformed by our method. We record timings for coherent (primary) and incoherent (indirect) ray distributions, separately. More specifically, for the primary rays all our scenes are rendered at 1920x1080 resolution and for each successful intersection with the input geometry, we randomly spawn 32 uniformly distributed rays around the triangle normal. Finally, we average our results over 2 distinct viewpoints and over 3 iterations. Since a BVH is commonly used as a query acceleration structure for instanced geometry in a two-level hierarchy, we also measure ray tracing performance, when emitting 32M uniformly distributed rays from the object's bounding box, directed inwards. This is presented in the next block of Figure 4 and effectively simulates how a leaf of the top-level hierarchy routes a query through a bottom-level BVH for arbitrary transformation of the object within a scene. Finally, the last block of the experiments first records average ray-bounding volume intersections for the case of SweepSAH-BVH. Average intersections account only for the indirect bounces, since they represent an incoherent workload. Additionally, we provide time-to-render measurements for LBVH and ATRBVH and for 1000 samples per pixel. We observe that there is a correlation between the number of bounding volume intersections and the expected ray tracing performance.

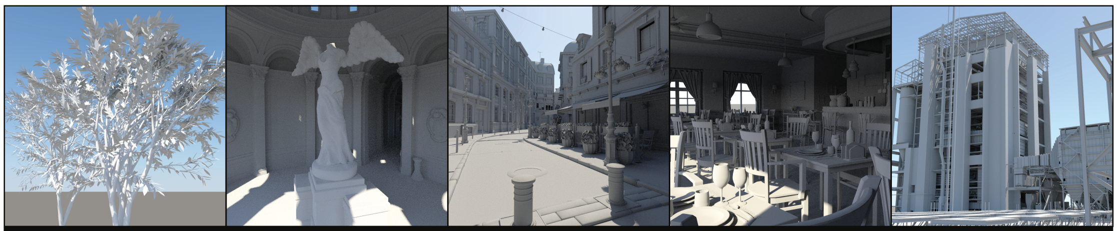
In Figure 5, we also present a visualisation of the number of bounding volume intersections during the traversal of primary rays on the same scenes, since primary rays are more easy to interpret, visually.

Our experiments include some fairly standard scenes as well as scenes that have traditionally benefited from orientation-aware hierarchies. For example, both the *Bistro Interior* and *Bistro Exterior* scenes where provided with no instance pre-alignment. The conversion to an OBB hierarchy is able to optimise the overfitted AABBs and create tight fitting bounds to the underlying geometry, resulting in a substantial performance improvement. In a similar manner, the *Powerplant* model has its own set of thin long geometry groups that also benefit from the OBB hierarchy generation. On the other hand, the temple scene is an example of a fairly well axis-aligned model that as a flat, non-hierarchical scene, does not significantly benefit from the OBB hierarchy although we did not measure serious performance degradation.

Moving on to the single object experiments, the converted OBB hierarchies on cases that are inherently expected to perform well, such as the *Hairball* and *Sheep* models, significantly outperform the initial AABB hierarchy in both primary and secondary ray workloads. This is also consistent with other results from the literature [WMZ*20, WBW*14]. The remaining models like *Trees*, *Dragon*, *Human* and *Crown* demonstrate the potential of OBBs to exploit orientation similarity at various levels of the hierarchy despite the purely spatial nature of the underlying builder.

For completeness, we also provide a breakdown of the conversion process into separate times for each step in Table 2. This includes timings for the construction of the initial LBVH and ATRBVH hierarchy of AABBs. The refitting step (see Section 3.4), which is the only step that needs to be executed in case of deformations, is the fastest.

Scene - # Triangles		Human - 1.2M	Crown - 3.5M	Sheep - 10.7M	Hairball - 2.8M	Dragon - 800K
		AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH
BVH conversion time		5.92	16.85	44.91	12.54	4.64
Time (ms) Primary rays	LBVH	1.07 / **0.96** (1.11x)	2.56 / **1.91** (1.33x)	147.82 / **15.25** (9.68x)	7.91 / **5.13** (1.54x)	1.23 / **0.91** (1.35x)
	ATRBVH	1.00 / **0.90** (1.10x)	1.81 / **1.42** (1.27x)	147.10 / **13.72** (10.71x)	6.46 / **4.20** (1.53x)	1.01 / **0.74** (1.36x)
	swpBVH	0.94 / **0.75** (1.25x)	1.88 / **1.35** (1.39x)	134.43 / **11.63** (11.55x)	6.30 / **4.01** (1.57x)	1.01 / **0.69** (1.46x)
Time (ms) Secondary rays	LBVH	3.06 / **1.92** (1.59x)	17.18 / **13.75** (1.24x)	518.08 / **41.95** (12.34x)	9.41 / **8.79** (1.07x)	5.64 / **4.34** (1.3x)
	ATRBVH	2.76 / **1.78** (1.54x)	13.39 / **10.69** (1.25x)	519.10 / **37.43** (13.86x)	10.90 / **7.80** (1.39x)	4.47 / **3.58** (1.24x)
	swpBVH	2.27 / **1.47** (1.54x)	12.66 / **9.66** (1.31x)	476.64 / **30.87** (15.44x)	7.11 / **6.81** (1.04x)	4.16 / **3.21** (1.29x)
Time (ms) Random BV rays	LBVH	1.64 / **1.59** (1.03x)	1.10 / **1.01** (1.09x)	127.82 / **19.77** (6.46x)	5.26 / **4.57** (1.14x)	1.64 / **1.46** (1.12x)
	ATRBVH	1.54 / **1.48** (1.04x)	0.94 / **0.84** (1.11x)	123.45 / **17.46** (7.06x)	5.06 / **3.71** (1.36x)	1.26 / **1.18** (1.06x)
	swpBVH	1.28 / **1.17** (1.09x)	0.90 / **0.77** (1.17x)	107.10 / **13.74** (7.79x)	4.17 / **3.41** (1.22x)	1.18 / **1.07** (1.10x)
Avg. intersections swpBVH		5.99 / **4.38** (1.36x)	8.53 / **6.05** (1.40x)	1282.80 / **128.36** (9.91x)	48.83 / **41.33** (1.18x)	15.15 / **12.01** (1.26x)
Time (sec.) to render	LBVH	2.11 / **1.75** (1.21x)	6.82 / **5.81** (1.17x)	268.81 / **25.86** (10.39x)	10.53 / **7.56** (1.39x)	2.56 / **1.93** (1.33x)
	ATRBVH	1.93 / **1.62** (1.19x)	5.06 / **4.07** (1.24x)	264.32 / **23.09** (11.45x)	8.81 / **6.34** (1.39x)	2.05 / **1.56** (1.32x)



Scene - # Triangles		Trees - 170K	Temple - 400K	Bistro exterior - 2.8M	Bistro interior - 1M	Powerplant - 12.7M
		AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH	AABB BVH / OBB BVH
BVH conversion time		0.97	1.98	12.78	4.84	52.63
Time (ms) Primary rays	LBVH	3.38 / **2.94** (1.14x)	**3.35** / 3.41 (0.98x)	7.74 / **5.37** (1.43x)	6.72 / **4.78** (1.40x)	5.91 / **4.01** (1.47x)
	ATRBVH	2.90 / **2.42** (1.19x)	**2.14** / 2.19 (0.97x)	5.12 / **4.05** (1.26x)	4.62 / **3.48** (1.32x)	3.26 / **2.58** (1.26x)
	swpBVH	3.29 / **2.44** (1.34x)	**1.76** / 1.83 (0.96x)	4.33 / **3.22** (1.34x)	4.77 / **3.15** (1.51x)	2.63 / **2.36** (1.11x)
Time (ms) Secondary rays	LBVH	15.22 / **12.89** (1.18x)	60.48 / **58.57** (1.03x)	148.09 / **131.96** (1.12x)	137.61 / **70.83** (1.94x)	29.03 / **26.67** (1.08x)
	ATRBVH	12.45 / **10.58** (1.17x)	**37.72** / 38.43 (0.98x)	117.72 / **96.04** (1.22x)	103.70 / **48.28** (2.14x)	21.01 / **17.66** (1.18x)
	swpBVH	12.12 / **10.08** (1.2x)	31.13 / **29.52** (1.05x)	91.13 / **72.57** (1.25x)	89.69 / **83.15** (1.07x)	18.63 / **13.98** (1.33x)
Time (ms) Random BV rays	LBVH	2.62 / **2.17** (1.20x)	- / -	- / -	- / -	- / -
	ATRBVH	2.15 / **1.77** (1.21x)	- / -	- / -	- / -	- / -
	swpBVH	1.95 / **1.72** (1.13x)	- / -	- / -	- / -	- / -
Avg. intersections swpBVH		33.73 / **25.03** (1.34x)	55.82 / **41.21** (1.35x)	94.97 / **58.75** (1.61x)	126.53 / **57.68** (2.19x)	32.99 / **24.11** (1.36x)
Time (sec.) to render	LBVH	6.79 / **5.60** (1.21x)	16.83 / **14.65** (1.15x)	33.69 / **29.49** (1.14x)	28.91 / **16.60** (1.74x)	14.15 / **11.18** (1.27x)
	ATRBVH	5.59 / **4.55** (1.23x)	10.48 / **9.52** (1.10x)	26.88 / **22.13** (1.21x)	21.22 / **11.79** (1.80x)	9.20 / **6.96** (1.32x)

Figure 4: *Build and traversal measurements for our transformed BVH hierarchy over the original AABB-based BVH. In the first row, we show the BVH conversion time (ms). Following rows present traversal performance (in ms) as measured for both coherent and incoherent rays. The last rows show average bounding volume intersections for swpBVH and time-to-render for* 1000 *spp.*

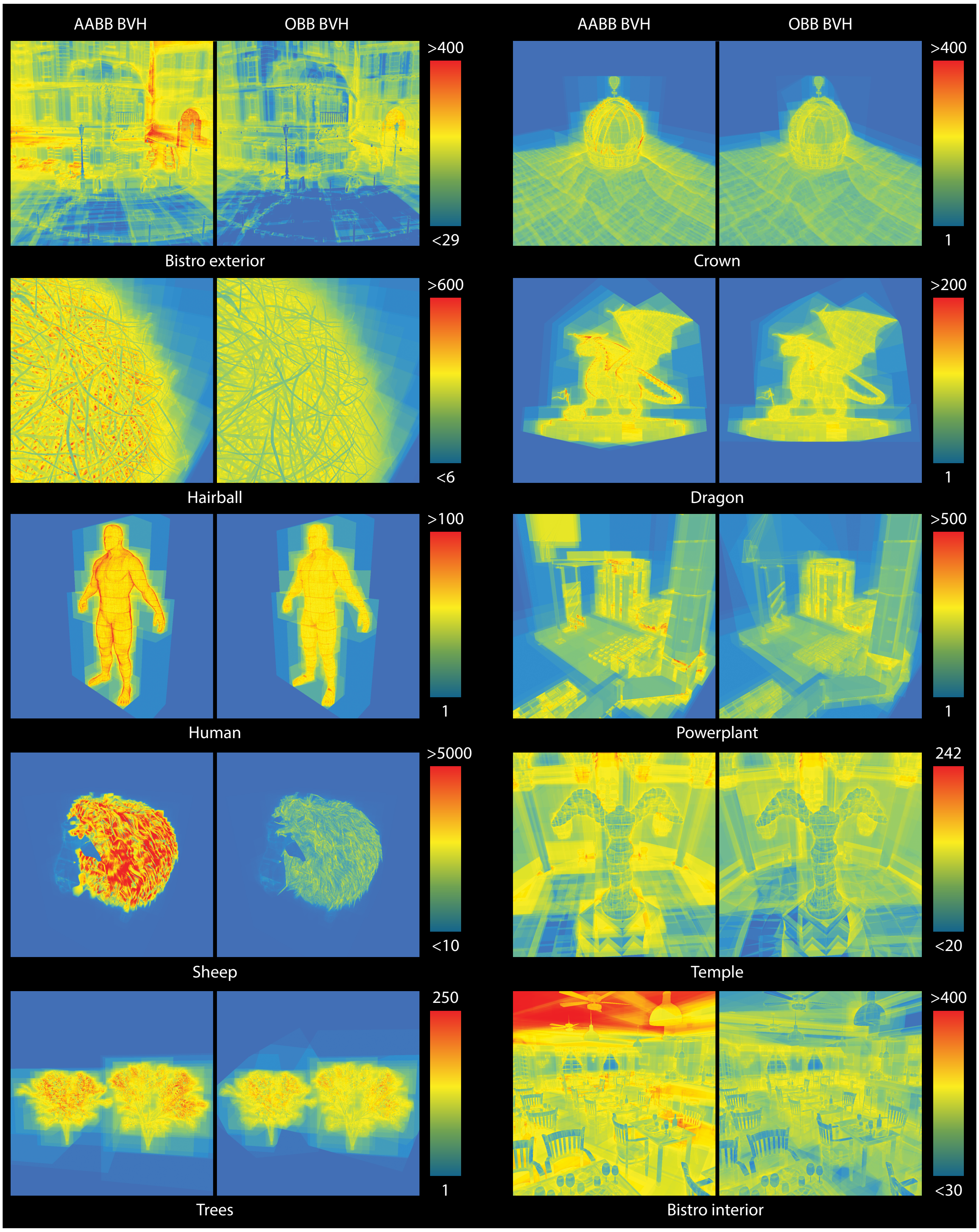


Figure 5: *Comparison of AABB/OBB hierarchy bounding volume intersections for primary rays. The AABB BVH was constructed using the SweepSAH-BVH builder.*

Scene	Builder		Conversion				Total construction time (relative)	
	LBVH	ATRBVH	Project	Select	Refit	Finalise	LBVH	ATRBVH
Human	5.56	10.96	3.28	0.84	1.15	0.66	11.48 (× 2.06)	16.89 (× 1.54)
Crown	12.94	25.36	8.97	1.92	4.18	1.78	29.80 (× 2.30)	42.22 (× 1.66)
Sheep	41.40	77.40	27.00	5.30	7.50	5.12	86.31 (× 2.08)	122.32 (× 1.58)
Hairball	12.01	22.62	7.36	1.59	2.14	1.45	24.55 (× 2.04)	35.15 (× 1.55)
Dragon	3.98	6.91	2.94	0.51	0.76	0.44	8.63 (× 2.17)	11.55 (× 1.67)
Trees	1.31	2.52	0.52	0.14	0.20	0.12	2.28 (× 1.74)	3.49 (× 1.38)
Temple	2.25	4.35	1.10	0.30	0.37	0.23	4.24 (× 1.88)	6.34 (× 1.46)
Bistro (exterior)	10.88	21.29	7.21	1.83	2.32	1.42	23.67 (× 2.18)	34.07 (× 1.60)
Bistro (interior)	4.50	8.56	2.72	0.70	0.88	0.54	9.35 (× 2.08)	13.41 (× 1.57)
Powerplant	41.44	79.56	29.20	8.91	8.69	5.83	94.07 (× 2.27)	132.19 (× 1.66)

Table 2: *BVH transformation time measured separately for each step and scenes presented in Figure 4. Columns indicate the time in milliseconds for each step execution.*

4.3. Rotated Scenes

A thorough study by Aila et al. [AKL13] challenges the correlation of theoretical quality metrics with actual ray tracing performance. Among other findings, they raised their concerns about performance degradation for ray traversal on non-axis-aligned geometry, the worst being scenes rotated by 45° over the diagonal axis $\{1,1,1\}$.

One advantage of OBBs is their inherit robustness to orientation change. However, since the OBB tree in our case uses the structure and primitive clustering optimised by an AABB-based builder, it was interesting to see to what degree the generated OBB trees retain this benefit, under the effect of rotation. To simulate this scenario, we applied a 45° rotation over the $\{1,1,1\}$ axis to our flat, non-hierarchical scenes and measured performance for ray traversal on the GPU, for the same experimental setup discussed in Section 4.2.

Our benchmarks on primary ray distributions, indicate an average relative improvement over the AABB-based BVH of 5.6×, 3.88× and 3.49× for the LBVH, ATRBVH and SweepSAH-BVH builders respectively. Additionally, for secondary rays, there is a similar performance improvement of 2.63×, 3.24× and 2.7×, respectively. This highlights the potential of our approach to remove a purely technical concern from the artists' creative workflow, that of axis alignment of scene geometry.

4.4. Memory Consumption

The additional storage in GPU memory required for our algorithm during the hierarchy construction of an input BVH with N leaf nodes, is a total of $(2N-1) \times 28$, 32-bit numerical precision values. This encompasses 14 floats for the min and max projection and another 14 integers for the corresponding indices of the associated vertices (see Section 3.3).

During traversal of the transformed hierarchy and considering the node layout we discussed in Section 3.5, the 2 AABB entries (12 floats) need to be replaced with two 4×3 transformation matrices (24 floats).

5. Conclusion and Future Work

We presented a method for the construction of an OBB hierarchy starting from any bounding volume hierarchy that requires minimal changes to the standard traversal algorithm. It can act as a post-processing step to existing binary BVH builders and is generic enough to naturally work with wide BVHs [YKL17] as well. Hierarchies based on OBBs have the potential to dispense with concerns regarding asset orientation in scene design and special handling of long, thin primitives or geometry groups. In a similar manner to how efficient hardware instance transformations of modern ray tracing APIs are already exploited in two-level hierarchies to optimise intersections with arbitrarily oriented long primitives [WMZ*20]. Both our fast OBB estimator and BVH converter can generalise this by providing the bottom level OBBs for primitive clusters and bottom-level acceleration data structures, respectively.

To further help the general graphics community, we provide full source code of our reference implementation at **https://github.com/cgaueb/obvh**.

Limitations. In this work, we reuse the output node structure imposed by the initial builder. Presently, this has the side effect that node linkage is decided using strictly spatial clustering criteria that optimise and collapse AABBs but do not necessarily contribute to efficient OBB generation for the clusters. In fact, in certain corner cases, these have a negative impact.

Another drawback of our approach is the increased temporary memory allocation for the intermediate storage of the min-max projections for each node of the hierarchy during construction. Additionally, the storage required for an OBB is larger than that for an AABB, affecting the final memory footprint of the hierarchy and potentially impacting the cache effectiveness, during tree traversal, albeit not in a noticeable manner. OBB compression schemes have been proposed [WBW*14], but the benefits of such approaches in the context of a full OBB hierarchy need further investigation.

Future work. In OBB hierarchies, ray transformations are performed at every node traversal step and, therefore, constitute a frequent computation. It would be worth investigating whether hardware ray transformation units could be employed for this operation. Unfortunately, access to this feature is not directly exposed in any of the current ray tracing APIs. Further investigation is also required regarding the choice of K for DiTO. We chose $K = 7$, as per authors advice, but different choices could show similar performance benefits with lower memory pressure.

Acknowledgements

The *Sheep* model was extracted from the Blender Foundation's demo scene repository. The *Hairball*, *Crown* and *Powerplant* models are from Morgan McGuire's resource archive. The Amazon Lumberyard *Bistro* and *Temple* scenes are part of the Open Research Content Archive. Finally, the *Dragon* model was taken from Benedikt Bitterli's Rendering Resources site. This research was funded by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "3rd Call for H.F.R.I. Research Projects to support Post-Doctoral Researchers" (Project No: 7310).

References

[AKL13] AILA T., KARRAS T., LAINE S.: On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, Association for Computing Machinery, p. 101–107. 9

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, p. 145–149. 5

[AM00] ASSARSSON U., MOLLER T.: Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools 5*, 1 (2000), 9–22. 2

[Ape14] APETREI C.: Fast and Simple Agglomerative LBVH Construction. In *Computer Graphics and Visual Computing (CGVC)* (2014), Borgo R., Tang W., (Eds.), The Eurographics Association. 2

[AW10] ABDI H., WILLIAMS L. J.: Principal component analysis. *WIREs Computational Statistics 2*, 4 (2010), 433–459. 2

[BHP01] BAREQUET G., HAR-PELED S.: Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *J. Algorithms 38*, 1 (jan 2001), 91–109. 2

[CGM11] CHANG C.-T., GORISSEN B., MELCHIOR S.: Fast oriented bounding box optimization on the rotation group $SO(3, \mathbb{R})$. *ACM Trans. Graph. 30*, 5 (oct 2011). 2

[CWK10] CHANG J.-W., WANG W., KIM M.-S.: Efficient collision detection using a dual obb-sphere bounding volume hierarchy. *Computer-Aided Design 42*, 1 (2010), 50–57. Advances in Geometric Modelling and Processing. 1

[DKKR09] DIMITROV D., KNAUER C., KRIEGEL K., ROTE G.: Bounds on the quality of the pca bounding boxes. *Comput. Geom. Theory Appl. 42*, 8 (oct 2009), 772–789. 3

[DP15] DOMINGUES L. R., PEDRINI H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In *Proceedings of the 7th Conference on High-Performance Graphics* (New York, NY, USA, 2015), HPG '15, Association for Computing Machinery, p. 13–20. 2, 4

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient bvh construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, Association for Computing Machinery, p. 81–88. 2

[GLM96] GOTTSCHALK S., LIN M. C., MANOCHA D.: Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, Association for Computing Machinery, p. 171–180. 2

[KA13] KARRAS T., AILA T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, Association for Computing Machinery, p. 89–99. 2

[Kar12] KARRAS T.: 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* (Goslar, DEU, 2012), EGGH-HPG'12, Eurographics Association, p. 33–37. 2, 5, 6

[KHM*98] KLOSOWSKI J., HELD M., MITCHELL J., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 21–36. 1

[LAM01] LEXT J., AKENINE-MÖLLER T.: Towards Rapid Reconstruction for Animated Ray Tracing. In *Eurographics 2001 - Short Presentations* (2001), Eurographics Association. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH Construction on GPUs. *Computer Graphics Forum* (2009). 2, 4

[LK11] LARSSON T., KÄLLBERG L.: Fast computation of tight-fitting oriented bounding boxes. *Game Engine Gems 2* (2011), 1. 2, 3

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (may 1990), 153–166. 6

[MB18a] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (2018), 1345–1353. 2, 5

[MB18b] MEISTER D., BITTNER J.: Parallel Reinsertion for Bounding Volume Hierarchy Optimization. *Computer Graphics Forum* (2018). 2

[MOB*21] MEISTER D., OGAKI S., BENTHIN C., DOYLE M. J., GUTHE M., BITTNER J.: A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum 40*, 2 (2021), 683–712. 2

[O'R85] O'ROURKE J.: Finding minimal enclosing boxes. *International Journal of Computer & Information Sciences 14* (1985), 183 – 199. 2, 3

[RW80] RUBIN S. M., WHITTED T.: A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph. 14*, 3 (jul 1980), 110–116. 2

[SAVBCNRM21] SABINO R., AUGUSTO VIDAL C., BENTO CAVALCANTE-NETO J., RODRIGUES MAIA J. G.: *Fast and Robust Ray/OBB Intersection Using the Lorentz Transformation*. Apress, Berkeley, CA, 2021, pp. 519–528. 5

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, Association for Computing Machinery, p. 7–13. 1, 4, 6

[TY09] TU C., YU L.: Research on collision detection algorithm based on aabb-obb bounding volume. In *2009 First International Workshop on Education Technology and Computer Science* (2009), vol. 1, pp. 331–333. 1

[WBW*14] WOOP S., BENTHIN C., WALD I., JOHNSON G. S., TABELLION E.: Exploiting local orientation similarity for efficient ray traversal of hair and fur. In *Proceedings of High Performance Graphics* (Goslar, DEU, 2014), HPG '14, Eurographics Association, p. 41–49. 2, 6, 9

[WMZ*20] WALD I., MORRICAL N., ZELLMANN S., MA L., USHER W., HUANG T., PASCUCCI V.: Using hardware ray transforms to accelerate ray/primitive intersections for long, thin primitive types. *Proc. ACM Comput. Graph. Interact. Tech. 3*, 2 (aug 2020). 3, 6, 9

[YKL17] YLITIE H., KARRAS T., LAINE S.: Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, Association for Computing Machinery. 9