





Stochastic Subsets for BVH Construction

L. Tessari¹  and A. Dittebrand^{†1,2}  M. J. Doyle¹  C. Benthin¹ 

¹Intel Corporation

²Karlsruhe Institute of Technology

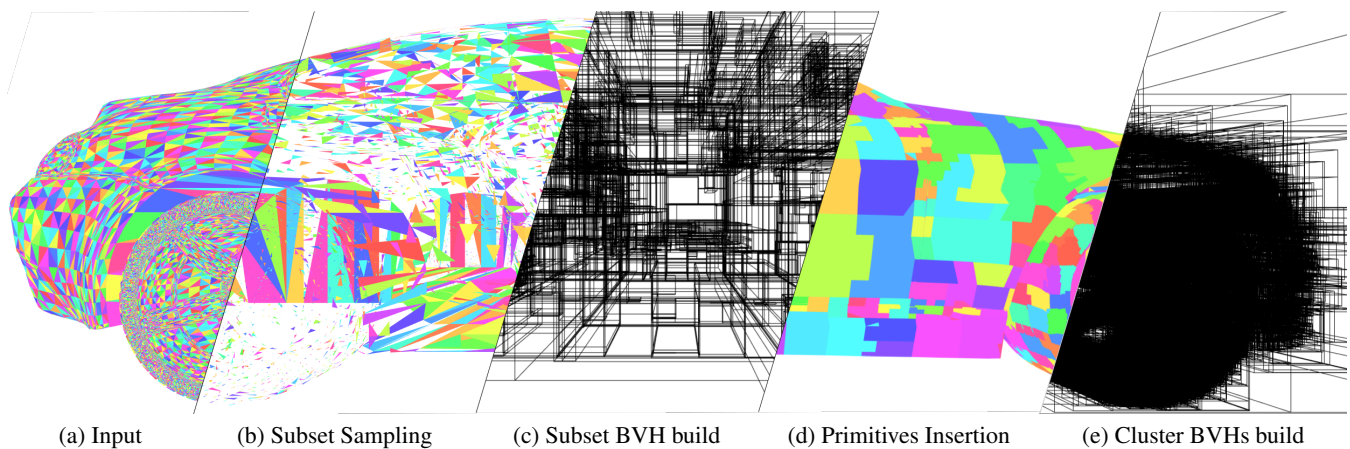


Figure 1: Overview of our proposed method. From the (a) input primitives, we (b) generate a subset, over which we (c) construct an initial BVH topology using an existing BVH constructor (a top-down builder for example). This step benefits from the smaller primitive count. The remaining primitives are (d) inserted into the leaves. We then (e) continue the construction process for each leaf.

Abstract

BVH construction is a critical component of real-time and interactive ray-tracing systems. However, BVH construction can be both compute and bandwidth intensive, especially when a large degree of dynamic geometry is present. Different build algorithms vary substantially in the traversal performance that they produce, making high quality construction algorithms desirable. However, high quality algorithms, such as top-down construction, are typically more expensive, limiting their benefit in real-time and interactive contexts. One particular challenge of high quality top-down construction algorithms is that the large working set at the top of the tree can make constructing these levels bandwidth-intensive, due to $O(n \log(n))$ complexity, limited cache locality, and less dense compute at these levels. To address this limitation, we propose a novel stochastic approach to GPU BVH construction that selects a representative subset to build the upper levels of the tree. As a second pass, the remaining primitives are clustered around the BVH leaves and further processed into a complete BVH. We show that our novel approach significantly reduces the construction time of top-down GPU BVH builders by a factor up to $1.8\times$, while achieving competitive rendering performance in most cases, and exceeding the performance in others.

1. Introduction

Bounding Volume Hierarchy (BVH) construction can easily become a bottleneck for real-time and interactive applications, especially where large quantities of dynamic geometry are involved. To offset the build cost, which can represent a substantial fraction of

the frame time, the tree is often refitted for many animation frames before being periodically rebuilt [LYMT06]. However, for highly dynamic geometry, the use of refitting can significantly compromise rendering performance due to lower BVH quality, until the BVH can once again be rebuilt.

The great variety of existing BVH construction algorithms either provide high traversal efficiency but slower construction speed, or fast construction speed and slower traversal efficiency. Some algo-

[†] Joint first authors

rithms make a compromise somewhere in between. An ideal algorithm would possess both properties, and would allow higher quality BVHs to be maintained under real-time dynamic conditions while minimizing the need for refitting. This would benefit both build and traversal.

One of the highest quality BVH build algorithms currently available is the *top-down* construction method [Wal07]. However, the drawback of this technique is that it currently cannot compete with the construction speed of lower quality algorithms. In this paper, we propose a new approach to top-down construction on the GPU which greatly accelerates its construction speed while preserving the same rendering performance in most cases, and exceeding it in others. We achieve this acceleration by reducing the number of primitives to be processed at key stages of the build. We leverage the fact that a carefully selected subset of the data gives a good approximation of the final BVH, especially concerning the upper topology which can exhibit a large working set in algorithms such as top-down construction. This strategy is motivated by a theoretical analysis, where we show how the split complexity can significantly speed up the construction. Our method proceeds in a number of stages, which are visualized in Figure 1. We first order the input primitives via a space-filling curve (a), and then maintain this ordering while building a Cumulative Distribution Function (CDF) to sample the dataset (b). Using stratified importance sampling allows us to select representative geometry that describes spatially and cost-wise the data distribution. After the approximated tree is built from this subset (c), we insert the remainder of the geometry by carefully selecting the optimal leaf where to insert each primitive (d). We then further process these leaves to produce the final complete BVH (e).

2. Related Work

Given the hitherto discussed importance of BVH construction speed and tree quality, a considerable variety of construction techniques have appeared in the literature. Some of the earliest high quality construction algorithms relied on a *top-down* approach to divisively cluster primitives [Wal07]. To the present day, these builders are still recognized as delivering among the best traversal performance [AKL13]. Later variants of top-down construction aim to improve and accelerate this general approach while retaining tree quality [GBDA15; GD16; HMB17]. *Bottom-up* approaches operate in the opposite manner and construct from the leaves to the root [WBKP08; GHFB13; MB18a; BDTD22]. *Linear Bounding Volume Hierarchy* (LBVH) techniques differ from both categories in that the BVH construction is transformed into a sorting process, and an implicit hierarchy is extracted [LGS*09; PL10; GPM11; Kar12; VBH17]. *Insertion-based* builders, while receiving little attention for many years [GS87], have re-emerged and work by incrementally inserting each primitive into an unfinished hierarchy [BHH13; MB18b]. Finally, *treelet restructuring* methods draw upon LBVH methods to an extent, while introducing a second step where independent subtrees in an initial LBVH are restructured or optimized to refine the hierarchy towards a globally more efficient hierarchy [KA13; DP15]. Other interesting approaches to BVH construction include *incremental construction* [BHH15] and construction via k-means clustering [MB16]. Fi-

nally, for a full overview of BVHs for ray tracing, we refer to the STAR report [MOB*21].

Our method builds on some of these existing algorithms, but introduces stochastic techniques as a main feature. Concerning previous work on stochastic methods applied to BVH construction, the only approaches known to us are a randomized plane splitting decision [NT03], and a metropolis-hasting chain to select which nodes to reinsert for animation-optimized T-SAH cost [BM15]. In contrast, we base all of our construction on estimates and stochastic processes.

3. Background

In this section, we give an overview of two topics of particular relevance to our novel BVH construction method. We first outline relevant techniques from the Monte Carlo and Quasi-Monte Carlo literature. For a more general view on stratification and importance sampling, we refer to [AP16]. Second, we provide an overview of binned SAH BVH construction [Wal07], as this is a major building block of our BVH construction method, and also represents our primary baseline for comparison.

3.1. Stratified Importance Sampling

Since our objective is to select a small and relevant part of the geometry representative of the whole scene, we can remap our problem into the sampling field and take advantage of its instruments. In this context, we can view our set of primitives as a 1D piece-wise function where we are interested in the single samples individually instead of the estimated value of its integrand. If each sample follows a meaningful distribution (i.e. according to area measure), we can obtain a representative subset of the data. To this end, we can leverage some known methods used in numerical integration. Standard Monte Carlo integration converges at a rate of $\mathcal{O}(\sqrt{n})$ in any dimension, without any restriction on the integrand smoothness. While this is a very desirable property, it is also apparent that it requires n^2 samples to halve the estimator error at any given point. In our case, we can see the variance as how well we can select our subset. *Stratification* and *importance sampling* are two variance reduction techniques that aim at improving this bound and that can be combined. Intuitively, if we can divide our integration domain into m different strata (each with a size of $s = 1/m$) and sample from each of those, we will have a better coverage of our function. In practice, it is often preferred to increase the number of strata progressively and take one sample in each. Low discrepancy sequences in Quasi-Monte Carlo, such as Sobol or Halton, can drastically increase the rate of convergence close to $\mathcal{O}(1/n)$ in many scenarios thanks to their sampling guarantees. For a more extensive discussion on random number generators, particularly regarding stratification and low discrepancy, we refer to [KGA*19; AP16]. Among each stratum, importance sampling can stir the evaluation to areas that are more relevant for our estimation. Given that random sequences are generated from a $[0, 1]^s$ distribution, importance sampling allows us to map them to our domain and reduce the overall variance. It is important to note that a bad fit could lead to infinite variance, so the mapping function must be chosen carefully.

3.2. Binned SAH BVH Construction

High quality BVH construction algorithms often follow a *top-down* approach. We begin with the root node containing the bounding box of the scene and divide it into two new child nodes, with each child node containing a disjoint subset of the primitives referenced by the parent node (this condition is relaxed in circumstances such as spatial splitting, but it is otherwise generally true). The AABBs of the child nodes are made to tightly bound the geometry contained inside them. The procedure can be repeated top-down for each new node created, until we reach the leaf nodes. Common criteria for when to create a leaf include the total primitive count, or a heuristic prediction of ray tracing efficiency.

Given this general flow of top-down construction, a critical question is how primitives should be distributed to the left and right child nodes. BVH traversal performance can vary greatly depending on how this decision is made. Simple methods, such as choosing the spatial median to allocate the primitives to the left and right child nodes, are inexpensive but do not achieve good BVH quality. By far, the most widely used approach when high quality trees are desired is the *Surface Area Heuristic (SAH)* [GS87; MB90]. The SAH can evaluate any valid node partition and return a cost that can be compared to choose the most advantageous candidate. The SAH cost C for partitioning a node representing a region (volume) in 3D space V into two child nodes L and R covering volumes V_L and V_R respectively can be expressed as:

$$C(V \rightarrow \{L, R\}) = C_T + C_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right)$$

where C_T and C_I are constants defining the cost of a traversal step and a primitive intersection respectively, SA is the surface area of a 3D volume, and N_L and N_R are the number of primitives contained in the left and right child nodes, respectively.

The SAH provides a way to compare different candidate partitions when dividing a node, but it does not tell us which candidates to evaluate. Any permutation which divides the primitives into two disjoint subsets with at least one primitive each constitutes a valid partition. For a large number of primitives, the number of such permutations will be astronomical, and many of these partitions will not be advantageous. We therefore need a way to select a manageable set of good partition candidates for comparison. A common way to achieve this is with the *binned SAH* approach. First, an axis of the node's AABB is chosen and a small number of axis-aligned planes (usually 16-32) are distributed along the node's extent (or, alternatively, the extent of the AABB which contains the centroids of the node's primitives). The regions between each splitting plane are referred to as "bins" and are used to define a histogram, with each bin tracking an AABB and a counter representing the extent and number of primitives in that bin. Primitives are allocated to each bin based on the position of their centroids. By sweeping over this histogram, we can evaluate the SAH cost for the child nodes that will be created by splitting the node with each candidate plane. We evaluate all planes and select the one with the lowest SAH cost.

4. Method

In this section we propose a new way to improve the construction speed of BVHs when using the top-down construction method, motivated by a theoretical analysis of its algorithmic complexity. Our approach to the problem is to avoid repeated access over all the primitives for each of the initial levels, achieved by reducing the starting size of the data that needs to be processed: by first building only over a subset of the primitives we can gain substantial performance improvements.

Our proposed method can be divided into four sequential steps:

- Subset sampling
- Subset BVH construction
- Primitives insertion
- Cluster BVHs construction

First, we generate a small representative subset of the primitives through stochastic importance sampling (*Subset Sampling*, Section 4.1). Then, from this subset, we construct an initial BVH (*Subset BVH construction*). For this, we can use a pre-existing BVH construction algorithm which we will refer to as the *interior builder*.

In the *Primitives insertion* step (Section 4.2), the remaining primitives are now inserted into the leaves of the subset BVH that effectively operate as clusters. Finally, we continue the BVH construction in parallel from each of these clusters (*cluster BVHs construction*) It is important to note that while in principle any builder can be used in this framework, $\mathcal{O}(n \log n)$ top-down builders like binned SAH will benefit the most. This is due to the higher bandwidth and compute demand from the upper part of the tree construction, which is mitigated by a smaller subset use. During the first step, we will also include a spatial reordering for the primitives according to a space-filling curve (i.e. Morton).

4.1. Subset Sampling

The selection of the subset has a large impact on the final topology. For example, naively picking the first m primitives will likely only cover a small part of the scene, and randomly would end up with an uneven distribution. Instead, we need to perform a meaningful choice across all primitives to obtain a representative selection. To this end, we leverage spatial ordering as well as stratified and weighted sampling. Stratification and spatial ordering ensure that we get a stratified selection of the primitives in space. Weighted sampling allows us to also steer the selection towards more important (larger) primitives, since it is beneficial to have them higher up in the tree to allow for more efficient early ray termination. If those were inserted only later in the insertion phase, they would end up in a lower level leaf of the tree, resulting in bigger bounding box overlaps and increased intersection costs.

The Subset Sampling step can be divided in four phases: primitives sorting (Section 4.1.1), primitives importance sampling (Section 4.1.2), weights clamping (Section 4.1.3) and some re-weighting to a more uniform distribution (Section 4.1.4). The two latter phases happen before the sampling and are meant as an algorithmic optimization rather than a theoretical improvement.

We use the primitives' bounding box diagonal as our sampling

weight to construct a CDF, from which we then sample m unique primitives. Some primitives might have a very large weight relative to the others, resulting in a very inefficient sampling by being selected multiple times. We consequently clamp weights to improve the process and ensure that important primitives will still be selected only once.

The unknown spatial distribution of the primitives can result in densely tessellated tiny regions that are under-sampled and thus create highly skewed primitives' distribution in the leaves of the Subset BVH. To alleviate this issue, we re-introduce some uniformity in the weights' distribution before sampling.

4.1.1. Primitives Sorting

Reordering the primitives following a space-filling curve has two main advantages: first, we can embed this form of spatial stratification inside our importance sampling procedure and second, we can exploit the data locality for access coherency and algorithmic optimizations. Since it naturally expresses a form of hierarchy in an ordered list, a contiguous subset of it forms a strata in space that we can exploit. We decided to use Morton sorting due to its implicit embedding of an octree structure in space[LGS*09], but in principle, a Hilbert, Moore or Peano curve could be used as well.

4.1.2. Primitives Importance Sampling

When selecting which primitives we want to use to create our approximate (Subset) BVH, we need to importance sample the ones that have a greater influence on its topology and SAH cost. In this context we aim to select large primitives whose bounding boxes would span considerably over space, instead of just sampling uniformly over the data. To this end, we can use a Cumulative Density Function (CDF) over a properly chosen measure and sample our primitives from it: appropriate measures would be the *primitive's area* or the *space diagonal's length* of the bounding box enclosing it. This approach guarantees that large bounding boxes stay at the top of the tree, where they can be evaluated early on. By means of stratified sampling, we can guarantee that a representative and well-distributed subset of the data will be selected. In many cases this can be enough, but we can offer even stronger properties: by keeping the same ordering on the CDF as the sorted primitives, we implicitly embed a spatial stratification of the geometry and reduce a 4D problem (3D space + measure) to 1D. This way we do not only follow the appointed distribution, but also reach out evenly over the whole geometry space, obtaining a Subset BVH that can better approximate the full one. We can see an example of our approach against unsorted uniform random sampling in Figure 2.

Generating Samples There are different ways to generate m stratified samples: one can be to naïvely divide the set into m strata and pick a sample inside each, and another could be using low discrepancy random sequences like Sobol. In the first case, if the weight of a single primitive (its value over the chosen measure) spans more than a single strata and thus can be selected multiple times, then the final subset can be arbitrarily smaller than m depending on the relative weight in the total CDF. In the second, the computation can be exceptionally long due to the amount of extra samples needed to reach m unique primitives. Note that this is true if the sequence is progressive, otherwise the fail case is similar to the naïve approach.

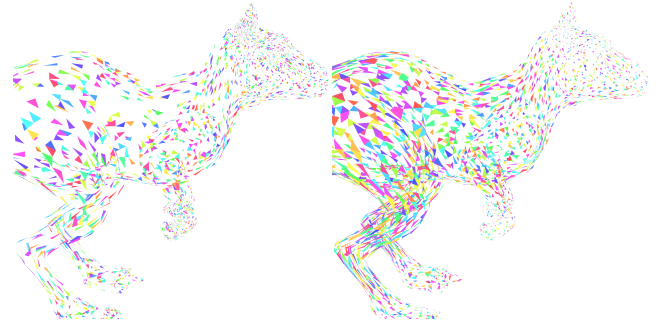


Figure 2: Naïve random sampling (left) vs our method (right), equal number of primitives: our importance sampling concentrates both spatially and on the larger triangles.

To overcome this issue, we will introduce a simple clamping technique in the next section.

4.1.3. Weight Clamping

Primitives with sampling probabilities larger than the normalized stratum size s (e.g. $s = 1/M$) are guaranteed to be sampled. However, large primitives can also cover numerous strata, resulting in many duplicate samples and a reduced efficiency of our sampling procedure. Prior to sampling the primitives, we clamp their weights such that the sampling probabilities of clamped primitives are equal to the stratum size. We define the clamped sampling probability of a primitive as

$$p_i = \frac{\min(w_i, c)}{\sum_j \min(w_j, c)}, \quad (1)$$

where i, j are primitives' indices, w is the sampling weight of a primitive and c is the clamping weight. To guarantee sampling of large primitives without duplicates, we only need to ensure that the sampling probability derived from the clamping weight itself is equal to the stratum size:

$$\frac{c}{\sum_j \min(w_j, c)} = s \quad (2)$$

Note that depending on the stratified random number generator, random points can be placed arbitrarily inside their strata. This means that consecutive points can have distances ranging from 0 to $2s$. In that case, a greater stratum size of $2s$ must be used for the clamping to still guarantee the sampling of large primitives. This also means that duplicates are inevitable even when using weight clamping - However, they are still limited to up to 3 duplicates per large primitive in the worst case.

Efficient Computation of the Clamping Weight We can see in Equation 2 that the sum changes depending on c , complicating a direct computation of c . The naïve approach would be to order the primitives by weights and to evaluate each weight as a possible clamping weight. In the following section, we present an efficient algorithm (Algorithm 1) requiring only one pass over the weights to compute an *approximate* clamping weight that is slightly larger than the optimal one. Equation 2 then turns into an inequality, i.e.

Algorithm 1: Histogram weight clamping

```

input : Array  $W$  filled with weights to clamp
         Stratum size  $s$  and bin base  $b$ , count  $b_c$  and offset  $o$ 
output: Clamping weight
1 hist  $\leftarrow [0 | i \in [0, b_c]]$ ; // histogram
2 for  $w \in W$  do // binning of weights
3   bin  $\leftarrow \min(\max(o + \lfloor \log_b w \rfloor, 0), b_c - 1)$ ;
4   hist[bin]  $\leftarrow$  hist[bin] + 1;
5 uSum  $\leftarrow 0$ ; // unclamped sum
6 cSum  $\leftarrow |W|$ ; // clamped sum
7 for  $i \leftarrow 0$  to  $b_c - 1$  do // bin search
8   clamp  $\leftarrow b^{i-o+1}$ ;
9   if clamp / (uSum + clamp  $\cdot$  cSum)  $\geq s$  then
10    return clamp;
11   uSum  $\leftarrow$  uSum + hist[ $i$ ]  $\cdot$  clamp;
12   cSum  $\leftarrow$  cSum - hist[ $i$ ];
13 return  $\infty$ ;

```

we only require the probability to be larger than s . We then find the lowest c that satisfies this constraint.

Our approach is to first build a distribution of weights through a histogram with b_c exponentially increasing bin ranges with base b (line 2). Mapping to the bin can be offset by the parameter o . We can then evaluate all boundaries between bins as potential clamping weights (line 7). The sum is approximated based on the collected distribution of weights before and after the boundary. We only track the number of weights that fall into a bin. While the clamped sum can be computed exactly, the unclamped sum is approximated through the upper bound of each preceding bin (line 11). In our implementation, we chose $b = \sqrt{2}$, $b_c = 128$ and $o = 64$.

This approach introduces two approximation errors: First, since we evaluate clamping weights only at bin boundaries, the resulting clamping weight can be off from the optimal one by up to a factor of b . Second, the overestimation of the unclamped sum can make the entire sum up to b times larger as well, therefore increasing the clamping weight by the same factor. The combined error bound is b^2 . We found a bin base of $\sqrt{2}$ to already be sufficient for our use case, resulting in an error bound of 2. For more details on the impact of using the clamping, we refer to the supplemental.

4.1.4. Uniformity

To ensure that densely tessellated regions are not underrepresented, we mix the clamped sampling probabilities with uniform probabilities through defensive importance sampling [Hes95]. Uniform sampling effectively increases the chance of sampling dense regions, and we can see its effect in Figure 3. To retain the sampling guarantee of large primitives, we need to perform a minor change to our weight clamping procedure.

We define our mixture probability p_i^* as

$$p_i^* = u \cdot \frac{1}{N} + (1 - u) \cdot \frac{\min(w_i, c)}{\sum_j \min(w_j, c)}, \quad (3)$$

where $u \in [0, 1]$ is the uniform fraction. Using this mixture di-

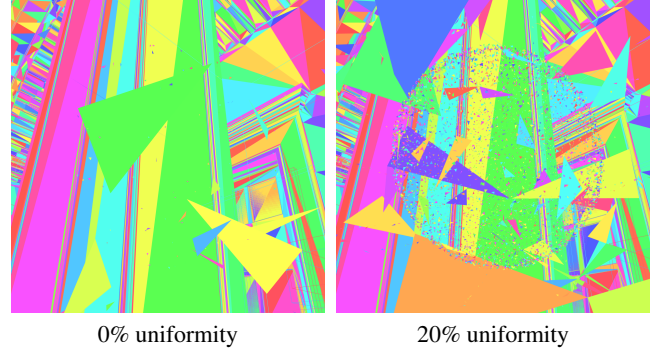


Figure 3: Impact of uniformity on sampling of tiny dense regions in scene San Miguel. Uniformity ensures that tiny dense regions remain well-represented in the subset.

rectly with the clamped probabilities would destroy the guarantee of sampling large primitives. Accordingly, we require the probability derived from the clamping weight to be equal to the stratum size:

$$u \cdot \frac{1}{N} + (1 - u) \cdot \frac{c}{\sum_j \min(w_j, c)} = s \quad (4)$$

After rearranging, we are left with:

$$\frac{c}{\sum_j \min(w_j, c)} = \frac{s - u/N}{1 - u} \quad (5)$$

This equation differs from equation 2 only in the right side. We can extract it as an updated stratum size s' :

$$s' = \frac{s - u/N}{1 - u} \quad (6)$$

s' can be used in place of s in the previous section when a uniform mixture is used when sampling. Intuitively, we increase the stratum size such that the resulting clamping weight reserves additional weight for large primitives. After applying the uniform mixture, this additional weight is then redistributed among all primitives, leaving large primitives with just enough weight to fully cover the actual stratum size s (therefore still being guaranteed to be sampled). The trade-off of increasingly adding uniformity is that, while densely tessellated regions are represented better, some large primitives start to lose the guarantee of being sampled, namely the ones whose weight is between the previous and new clamping weight. As such, the uniformity should not be set too high. We generally recommend a uniformity between 10% and 20%. For more details on the effect of adding uniformity we refer to the supplemental.

4.2. Primitives Insertion

After the subset BVH has been built using the interior builder, the remaining primitives need to be inserted into its leaves to continue the construction process. We will refer to these leaves as *clusters*. The problem definition of this phase is to associate each of the remaining primitives with a cluster. The insertion decisions are

guided by a cost model as detailed in Section 4.2.1. We minimize this cost model by evaluating multiple candidate leaves for insertion and selecting the one with the lowest cost. The algorithm is detailed in Section 4.2.2.

4.2.1. Cost Model

Our cost model is inspired by [BHH13]. They minimize the SAH cost of an existing BVH by performing topological rearrangements. Instead of computing the SAH directly for a possible rearrangement, they only compute its absolute change by subtracting the previous bounding box surface areas from the updated ones. Minimizing the change gives the same result as minimizing the SAH directly. However, it has the added advantage that only the affected nodes have to be considered (the change is zero for all other nodes). We apply this approach to our insertion decision in the same way, but we are not performing rearrangements to the topology. Instead, we only need to insert new primitives into the leaves of the current topology.

The flattened SAH metric [MB90] expresses the cost of a (sub)tree with root node N as

$$C(N) = \frac{1}{SA(N)} \left[C_T \sum_{N_i} SA(N_i) + C_I \sum_{N_i} SA(N_i) |N_i| \right] \quad (7)$$

where SA is the surface area of internal (N_i) or leaf (N_i) nodes, and C_T, C_I the traversal and primitive intersection costs respectively.

The cost increase of inserting a new primitive p in the tree can be formulated by the individual increases at a leaf (I_l) and internal node (I_i) level:

$$I(p, N) = \sum_{N_p} I_l(p, N_p) + I_i(p, N) \quad (8)$$

with

$$\begin{aligned} I_l(p, N) &= C_I SA(N'_l) |N_l + 1| - C_I SA(N_l) |N_l| \\ &= C_I ((SA(N'_l) - SA(N_l)) |N_l| + SA(N'_l)) \end{aligned} \quad (9)$$

N'_l is the union of the leaf N_l and the primitive's bounding box. Similarly, each node in the trail of parent nodes N_p arriving at the root from the leaf N_l , would increase their cost accordingly as

$$I_i(p, N_p) = C_T (SA(N'_p) - SA(N_p)), \quad (10)$$

where N'_p is the union of the inner node N_p and the primitive's bounding box.

4.2.2. Pruning Morton Window Search

Evaluating the cost function for each primitive in every cluster would be too expensive in practice. Similar to [MB18a], we leverage the already computed spatial ordering of primitives from the subset sampling phase (Section 4.1) in order to perform a localized search, in our case of insertion candidates (Figure 4). The algorithm considers a fixed number of subset primitives and their clusters around a primitive. Since subset primitives are sparsely represented in the original array, we store the cluster pointers in a separate compacted array (leafForSubsetPrimitive). We use a prefix sum

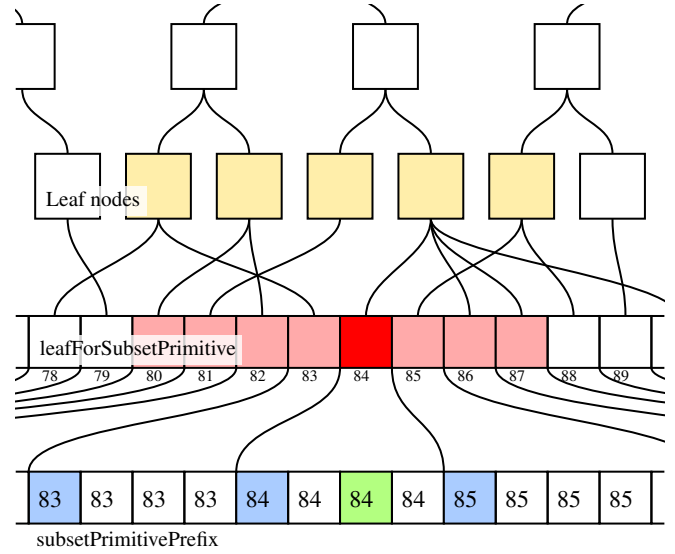


Figure 4: Window construction of the pruning Morton window search. Each primitive possesses a reference (subsetPrimitivePrefix) into a compacted array of subset primitives (leafForSubsetPrimitive). This array contains references to the leaf node (cluster) each subset primitive belongs to. For each primitive we take its reference (i.e. green) and then generate a window around its previous subset primitive in the compacted array (red). Each leaf node pointed to in the window is considered for insertion (yellow).

Algorithm 2: Pruning Morton window search

```

input : Primitive  $p$  with index  $i$  and window size  $w$ 
output: Leaf node to insert primitive into
1  $m \leftarrow \text{subsetPrimitivePrefix}[i]$ ;
2  $\text{minCost} \leftarrow \infty, \text{minLeaf} \leftarrow \perp$ ;
3 for  $j \in [m - w, m + w]$  do
4    $\text{leaf} \leftarrow \text{leafForSubsetPrimitive}[j]$ ;
5   if  $\text{leaf} = \text{minLeaf}$  then continue;
6    $\text{cost} \leftarrow I_l(p, \text{leaf});$  // Eq. 9
7    $\text{node} \leftarrow \text{getParent}(\text{leaf})$ ;
8   while  $\text{node} \neq \perp \wedge \text{cost} < \text{minCost}$  do
9      $\text{diffCost} \leftarrow I_i(p, \text{node});$  // Eq. 10
10    if  $\text{diffCost} = 0$  then break;
11     $\text{cost} \leftarrow \text{cost} + \text{diffCost}$ ;
12     $\text{node} \leftarrow \text{getParent}(\text{node})$ ;
13  if  $\text{cost} < \text{minCost}$  then
14     $\text{minCost} \leftarrow \text{cost}$ ;
15     $\text{minLeaf} \leftarrow \text{leaf}$ ;
16 return  $\text{minLeaf}$ ;

```

to perform this compaction (subsetPrimitivePrefix), which we also use to later index into this compacted array. Due to the sparsity of subset primitives, the search will cover a large spatial region even with small windows.

The actual search is detailed in Algorithm 2. We track the cluster with the smallest cost and iterate over the window. For each

candidate, we compute the cost by traversing the tree upwards towards the root. Our cost is simply the sum of surface area changes of the traversed inner nodes (Eq. 8). We exploit the fact that all cost terms are positive, i. e. while we traverse towards the parent, the cost increases monotonically. When the cost estimate of the current candidate exceeds the previous minimum, we can abort the traversal early. We additionally avoid revisiting the minimum node found so far (Line 5) and abort the traversal if the cost increase is zero (Line 10).

4.3. Algorithmic Complexity

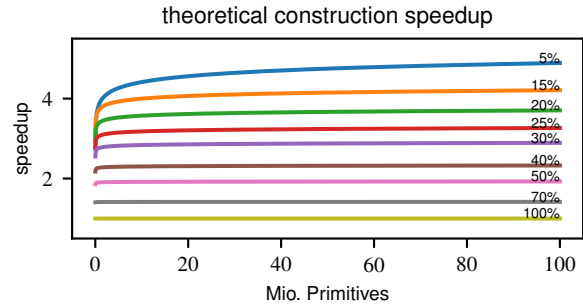
Different approaches to BVH construction give specific trade-offs between algorithmic complexity and final tree quality. Fast Morton builders usually come with a $\mathcal{O}(nk)$ complexity from their radix sort, where k is the key length and n the number of primitives, but suffer from lower SAH quality. Top-down builders, instead, have a more expensive $\mathcal{O}(n \log n)$ complexity, but give better SAH quality and thus faster intersection tests. In this section we will focus on the latter case.

We can break up our algorithmic complexity in three main components: the subset BVH creation (1), the cluster BVHs construction (2) and the extra steps required for the full build process, namely sorting, sampling and insertion that we will refer as overhead (3). In (1) we use only a subset M of the data, bringing the complexity to $m \log m$, with $m = |M|$; after the upper part is built, the cluster BVHs construction (2) needs to iterate over n primitives and each tree has on average n/m elements: this can be expressed as $n \log(n/m)$. Finally the overhead (3) of our method touches n primitives if the spatial sorting is enabled, m during the sampling and $n - m$ in the insertion step; in this case we utilize $\alpha \in [0, 1]$ as a factor relative to the construction time for our overhead and conservatively bound (3) to $\alpha n \log n$.

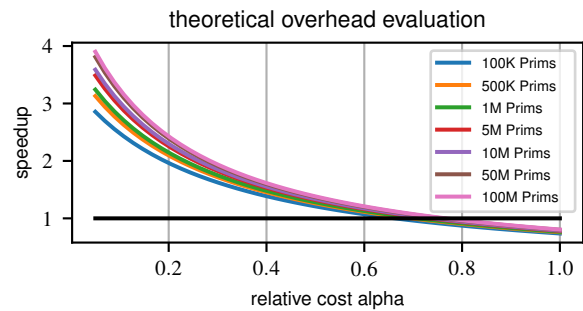
By considering the construction time alone, we can evaluate an upper bound on the speedup increase by the ratio between a standard top-down builder and the sum of our two construction steps (1,2) as $\frac{n \log n}{m \log m + n \log(n/m)}$. As shown in Figure 5a, depending on the amount of primitives used for the subset M , there is a wide theoretical speedup ranging from $1 \times$ to $5 \times$: while the trend is similar, almost plateauing after a few hundreds of thousand primitives, there is a slow but steady increase inversely proportional to the subset size. Since this amount also affects the final quality of the BVH, in the next experiment we will consider a reasonable size of 20% primitives, which already gives a theoretical $3.5 \times$ speedup.

When also adding the overhead (3) for bounding our theoretical gain, we need to consider the relative cost introduced compared to the pure building time: this results in the speedup ratio $\frac{n \log n}{m \log m + n \log(n/m) + \alpha n \log n}$. To this end, we can see how $\alpha \in [0, 1]$ behaves with different numbers of primitives ranging from tens of thousands to tens of millions (Figure 5b). As it is apparent, the overhead with 20% data needs to be at most 60% of the build time of a standard top-down builder; after that, any theoretical gain is lost. Another insight from this experiment suggests that our method is mostly independent of the number of primitives involved.

To summarize, with a baseline of 20% data for M ($m = 0.2 \cdot n$) and 10% overhead cost (alpha), we have seen a theoretical gain



(a)



(b)

Figure 5: Theoretical analysis of construction time speed-up relative to a top-down builder: the upper bound gain by different sized subsets in the construction time without overhead (a) is readily reached after a few hundreds of thousand primitives, and increases very slowly afterwards. With overhead and a 20% subset size (b), the theoretical upper bound gain decreases and breaks even when the extra compute reaches 60% of the construction time, without much variation with an increasing number of primitives (colored lines, from 10^5 to 10^8).

of up to $3 \times$ by using our approach against a standard top-down builder. This result holds true as long as the builders have similar performances. Given different hardware and software implementations, however, this assumption might not hold and there could be other factors to consider. We will analyze how this upper bound relates to our tests in our evaluation section 6.4.

5. Implementation

Terminology In the following, we will use the terminology from OpenCL to refer to the GPU programming model. A work group (CUDA: Thread block) is a set of threads that can directly synchronize with each other and also exchange data through fast but limited shared memory. Work groups are further subdivided into sub groups (CUDA: Warp). Threads inside a sub group are executed in lockstep and can directly exchange data through register permutations.

Framework We implemented our algorithm both as a GPU-builder based on oneAPI DPC++ (<https://www.oneapi.io/>) and a CPU-builder in PBRT [PJH16]. The GPU-builder implements the presented method from Section 4, while the CPU builder was used to test various approaches that led to the final design (We refer to the supplemental for the analysis of various parameters of our builder on the CPU). In this section we focus on the implementation details of the GPU-builder.

Interior Builder We use a binned SAH builder in the style of Wald [Wal07] as the interior top-down BVH builder. Each axis is considered for splitting using 16 bins each. We employ their horizontal and vertical parallelization technique with an additional middle phase to efficiently exploit the massive parallelism of the GPU at each BVH level. Since nodes are initially few, the horizontal phase is parallelized over primitives. For each iteration (seven in total), three kernels are launched for (1) initializing and (2) accumulating bins and (3) computing the splits and partitioning primitives. Only nodes with primitive counts larger than the average over all active nodes are considered, leading to a more even primitive distribution for the following phases. After the horizontal phase, we transition to the middle phase where active nodes with more than 1024 primitives are processed. For each iteration (twelve in total), a single kernel is launched where each work group processes a single node. The vertical phase then proceeds to compute the remaining subtrees of all nodes. This is performed with a single kernel launch. Each sub group in the kernel processes a single node.

For building cluster BVHs, we directly use vertical parallelization, since the number of clusters typically already far exceeds the number of concurrently executing sub groups. This is also the reason why uniformity (Section 4.1.4) must be added to the subset sampling. Otherwise, large clusters of highly tessellated geometry will introduce a strong load imbalance, hindering individual sub group processing capabilities and arbitrarily increasing the overall build time.

Subset Sampling We use single-precision floating point numbers for the CDF for performance reasons. While the available precision in large scenes is not enough to faithfully represent regions with low probability, we found that it did not affect the results much. However, rounding errors in the CDF do have an effect. The prefix scan routine which we employ performs the scan across an implicit hierarchy with three levels. First, a scan per sub group is performed, then a scan per work group, and finally a scan over all work groups. This approach exhibits good error properties, since it is comparable to a pairwise summation (but wider).

The random numbers used during the sampling stem from equidistant points with a distance of the stratum size s in $[0, 1]$ with a random initial offset. The sampling quality was not affected much compared to Sobol points. However, the highly coherent access during the bisection of the CDF led to a $5\times$ performance increase in the sampling dispatch. Additionally, the equidistant spacing decreases the chances of duplicates (caused by the approximate weight clamping) with the trade-off in a potentially smaller subset size.

Primitives Insertion We performed slight modifications to the pruning Morton window search (Algorithm 2) to better exploit the parallelism of the hardware. Instead of having each thread traverse its own window, we unify the windows of all threads in a sub group. All threads then proceed to step through this larger shared window in lockstep, which results in perfectly coherent memory access. Additionally, it seems beneficial to first evaluate the center of the window for each thread independently. While the memory access is relatively inefficient in that case, the found node is oftentimes already the optimum. As such, pruning may be triggered early when the window is traversed. With both optimizations, the performance of the search dispatch improved by 15%.

Memory Requirements Compared to the binned SAH builder, our stochastic builder additionally needs 32 bytes of memory for each of the N primitives (Morton codes, CDF, subset mask & prefix sum and insertion selection) and 44 bytes for each subset primitive (Compacted primitive bounding boxes, back-references to nodes as well as atomic counters for the insertion step). None of this memory is used when the cluster BVH build phase starts, so parts or even all of the memory can be aliased with memory required by the binned SAH builder. For example, the preallocated node memory can be used, since only the smaller subset BVH occupies it by that point. In the scene Crown (Fig. 6), we measured a memory consumption (binary BVH construction only) of 411 MiB, compared to 131 MiB of the binned SAH builder. Other builders detailed in Section 6.1 require 177 MiB (LBVH), 242 MiB (PLOC++) and 196 MiB (ATR BVH). Note that all memory is allocated upfront in our implementation.

6. Evaluation

In this section, we analyze our results from multiple perspectives. First, we compare our method's build performance with other GPU builders (Section 6.1). Second, we examine the variance of our approach due to stochastic sampling (Section 6.2). Third, we provide a break-down of our performance compared with a top-down binned BVH builder (Section 6.3). Finally, we examine how our theoretical complexity analysis matches our results (Section 6.4).

We evaluate our GPU builder on Intel Alchemist A770 GPU [Cor22] (32 Xe cores). The remaining system consists of an Intel Core i5 9600K CPU clocked at 3.70GHz with 16GB of DDR4 RAM running Ubuntu 20.04 LTS Linux OS on an NVMe SSD.

We render all images at a resolution of 1024×1024 using primary rays at 1spp and ambient occlusion with 64 indirect rays using the six scenes presented in Figure 6. The scenes vary in primitive count from 279K (Crytek Sponza) to 7.9M (San Miguel).

Our builder is, unless otherwise stated, parameterized by a uniform fraction of 10%. We parameterize the subset size as a fraction of the total primitive count in a scene (Subset Fraction) and is set by default to 20%.

6.1. Build Performance

We compare our method against existing build algorithms targeted at GPUs, namely LBVH [LYMT06], PLOC++ [BDTD22], ATR-BVH [DP15] and binned SAH construction [Wal07]. We base

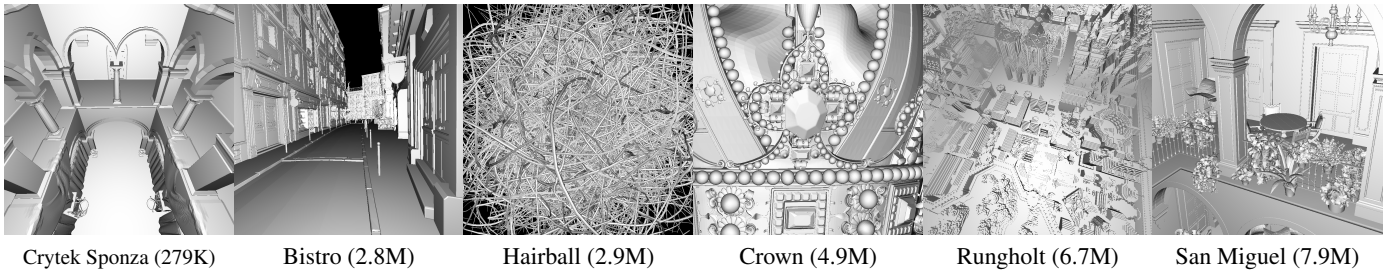


Figure 6: Overview of scenes and camera views we tested with. All scenes were taken from [McG17]. The primitive count is given in the parentheses.

	LBVH	PLOC++	ATRBVH	Stochastic (ours)	Binned SAH	LBVH	PLOC++	ATRBVH	Stochastic (ours)	Binned SAH
	Crytek Sponza					Bistro				
Host build time (ms)	2.25	6.14	4.97	10.47	9.71	9.61	14.22	20.63	20.0	23.24
Device build time (ms)	1.49	2.28	3.79	7.14	7.69	6.11	7.45	19.25	16.34	20.78
SAH cost	61.32	47.47	51.47	42.7	50.01	74.53	53.75	62.2	38.43	39.75
Primitive throughput (MPrim/s)	63.21	23.19	56.16	13.6	14.66	151.76	102.54	137.01	72.91	62.76
Primary (GRay/s)	2.745	3.479	3.314	3.124	2.83	1.276	1.349	1.302	1.38	1.367
Ambient occlusion (GRay/s)	2.844	3.187	2.987	3.09	2.846	0.814	0.882	0.813	0.883	0.849
	Hairball					Crown				
Host build time (ms)	7.0	12.24	19.45	18.03	23.75	10.58	18.18	30.9	25.37	34.78
Device build time (ms)	5.96	6.51	18.19	14.44	21.61	9.45	11.03	29.61	21.3	32.16
SAH cost	320.11	283.82	301.96	184.52	182.3	22.15	19.64	19.69	14.82	15.39
Primitive throughput (MPrim/s)	205.83	117.66	148.04	79.85	60.63	231.0	134.42	157.56	96.35	70.27
Primary (GRay/s)	1.255	1.312	1.305	1.362	1.365	2.38	2.509	2.522	2.635	2.651
Ambient occlusion (GRay/s)	1.089	1.096	1.11	1.184	1.194	1.928	2.01	2.001	2.102	2.113
	Rungholt					San Miguel				
Host build time (ms)	13.41	15.35	42.08	30.08	42.69	18.08	26.74	56.23	40.28	73.47
Device build time (ms)	12.33	10.58	40.77	25.84	40.26	16.92	17.97	54.81	36.26	70.93
SAH cost	130.12	90.4	84.95	69.97	62.74	64.35	41.44	47.17	38.81	40.25
Primitive throughput (MPrim/s)	250.01	218.44	159.34	111.45	78.52	250.19	169.13	140.14	112.29	61.56
Primary (GRay/s)	2.423	2.739	2.939	2.86	2.967	1.789	2.401	2.299	2.369	2.302
Ambient occlusion (GRay/s)	3.218	3.489	3.813	3.652	3.964	1.106	1.559	1.395	1.466	1.459

Table 1: Comparison of our stochastic builder with existing GPU build algorithms on Intel Alchemist A770 GPU (32 Xe cores) using Ubuntu 20.04 Linux.

our LBVH and PLOC++ implementation on the work of Karas [Kar12] and Benthin [BDTD22] respectively, while we ported the publicly available code for ATRBVH (<https://github.com/leonardo-domingues/atrbvh>) into oneAPI DPC++. The implementation of the binned SAH builder is identical to the interior builder we use for the subset BVH (Section 5).

We measure ray tracing performance using hardware traversal. For the binned SAH and our stochastic builder, we stop the construction process of the BVH as soon as there are eight or fewer primitives in a node. LBVH, PLOC++ and ATRBVH construct hierarchies with one primitive per leaf. All implementations build binary BVHs, that are then converted to the hardware format on GPU.

As the expected input format is quads, primitives are converted prior to construction, thus roughly halving the effective number. Host time includes time spent on device, dispatches and synchronizations on CPU. Conversion times from binary BVH (BVH2) to hardware specific BVH format (HW BVH) are not included. For a recent study on the traversal performance and SAH cost impact of different BVH formats after conversion from BVH2, we refer to [MB22].

We summarize our findings in Table 1. Compared to the binned SAH builder, we improved build times by $1.33\times$ on average ($1.47\times$ on device time). We generally observe that for large scenes the build time reduction is more significant. With San Miguel (7.9M

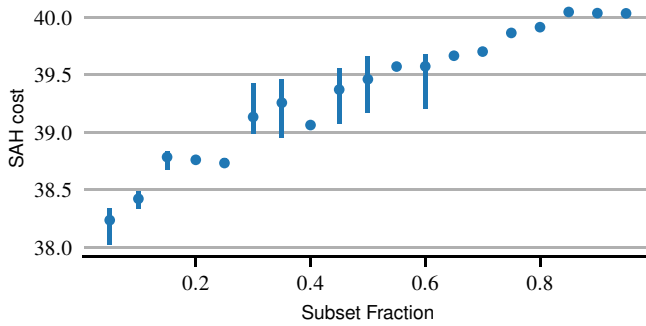


Figure 7: SAH variance on the San Miguel scene at different subset fractions (5% to 95%) using 50 random seeds each. We can see how it stays stable and slowly converges to the higher SAH of the reference binned SAH value when increasing the subset fraction.

primitives), we reach an improvement of $1.82\times$ ($1.96\times$ on device), while only on Crytek Sponza (279K primitives) we see a small increase in the host build time (7%). The additional dispatches and synchronization time we introduce are not amortized in that case. The stochastic builder is competitive with ATRBVH in all cases involving more than a few hundred thousand primitives. At the same time, while not reaching similar build time competitiveness, it considerably reduces the well-known gap from top-down builders to faster ones like LBVH or PLOC++. In terms of SAH quality, our stochastic builder is able to maintain a comparable or even lower cost than the binned SAH builder. As a result, both builders are consistently the lowest in our tests. Although this consistency does not translate linearly to the final rendering performance due to the hardware format conversion, our stochastic builder still remains the best or second-to-best and within $\pm 1\%$ from the more expensive binned SAH in all the scenes, with the exception of Rungholt.

6.2. Variance Analysis

An essential question for our approach is: how robust and reliable is it against a simple change in the subset due to different random numbers? This can happen for many reasons, such as different rng implementations, different initialization seeds, different floating point precision and so on. Another important implication is how this relates to the subset size: does some subset have higher variance at a specific threshold, or does it remain stable overall? To answer this question, we can look at Figure 7. This test was run on the San Miguel scene, with 50 different seeds in each step. Overall the SAH cost stays very stable, with an occasional spike of less than 1%, and tends to decrease when lowering the subset fraction. In other scenes, we observe the contrary trend, although the deviations remain in a similar range. During our tests, we noticed that sizes of around 15-20% primitives for the subset give good trade-offs between host time build and SAH quality/performance. For a more detailed analysis over multiple scenes, we refer to the supplemental.

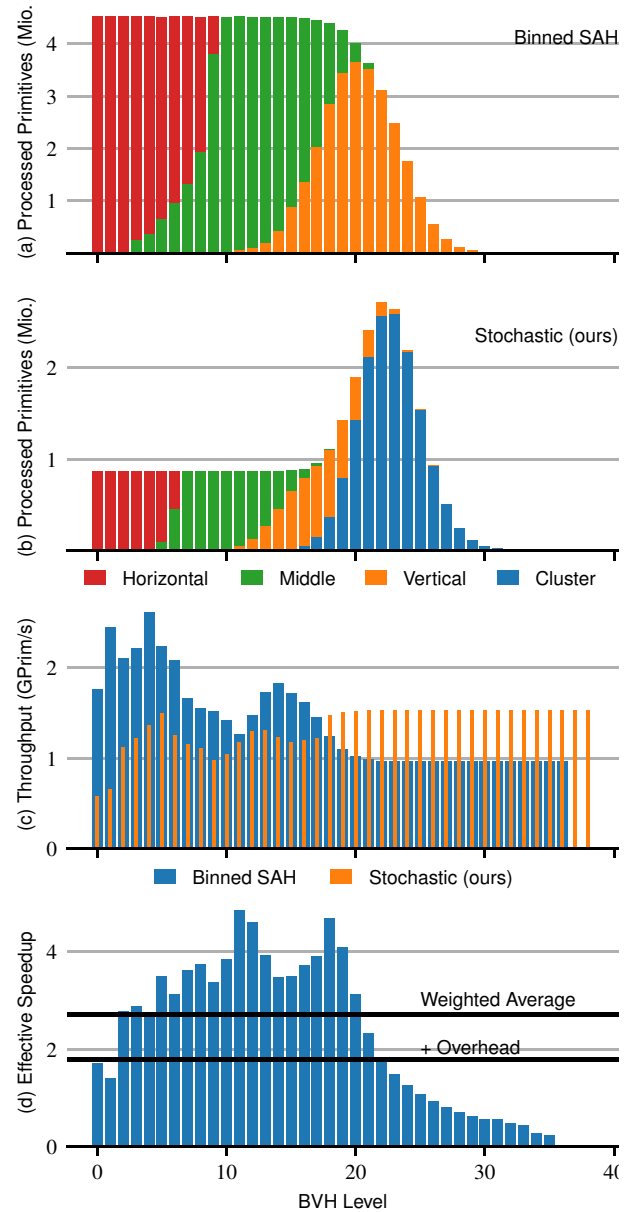


Figure 8: Per-level comparison of our stochastic builder (20% data) with the binned SAH builder in scene San Miguel. For the first few levels, our builder accesses fewer primitives (a, b). Although the effective throughput in the first levels is considerably lower (c), the effective speedup (d) due to the reduced primitive set is still $2\text{-}4\times$ in the stochastic BVH construction (first ~ 20 levels). On average, the speedup is around $2.7\times$, or $1.9\times$ when including our additional overhead. Note that the topologies are not identical, hence the additional levels of our builder.

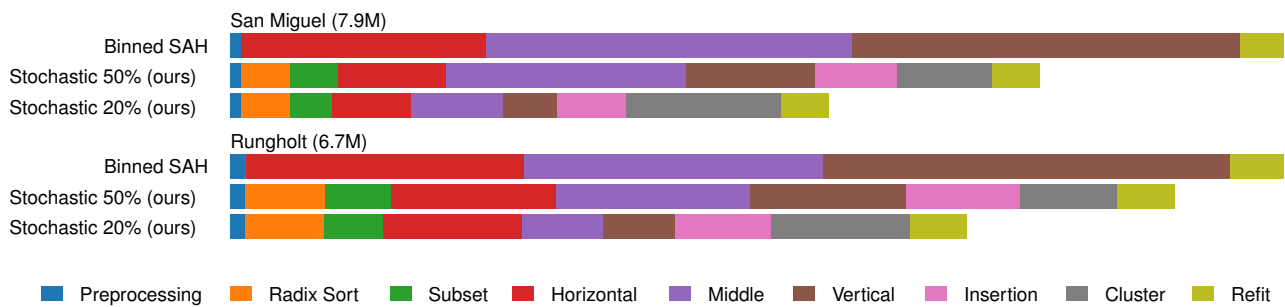


Figure 9: Relative timing breakdown of our stochastic builder and the binned SAH builder in the San Miguel and Rungholt scenes. Our additional overhead at a subset fraction of 20% amounts to 27% and 32%, respectively, while we observe a total build time reduction of 57% and 70%. With increasing subset size the middle phase shows the highest increase in time. For a description of the horizontal, middle and vertical pass we refer to Section 5.

6.3. Build Time Breakdown

Our stochastic build consists of many passes contributing to the total running time. Figure 9 shows a breakdown of the individual passes at a subset fraction of 20% and 50% in the San Miguel and Rungholt scenes. We also show a breakdown of the Binned SAH builder. While the added overhead is non-negligible, it is more than amortized by the time reduction of the horizontal, middle and vertical phases that exist in both algorithms at 20% subset fraction (for a description of these passes, see Section 5, *Interior Builder*). At 50%, we can see a significant increase in the middle phase execution time. In general, the efficiency of subset BVH construction does not scale linearly with a reduction in the other phases or with the SAH cost, suggesting that a carefully chosen small subset of geometry is able to achieve good performance compared to a more sizeable one at a fraction of the cost.

We additionally present a primitive throughput analysis on the individual levels of the BVH for our stochastic builder and the binned SAH builder in the scene San Miguel (Figure 8). We collect histograms per dispatch on how many primitives are processed for each level of the BVH and then proportionally distribute the execution time of the dispatch to each level. The timings are more fine-grained for the first levels, since those are processed by multiple dispatches. We found that the primitive throughput in the first levels, where construction occurs exclusively based on the subset, differs by a factor of $0.4\times$ compared to the binned SAH builder.

This discrepancy is explained by the scaling behaviour of the horizontally-parallelized build phases. We suspect that the constant overhead due to bin accumulation and evaluation becomes a major factor with lower primitive counts. Reducing this overhead is therefore an interesting area for future work.

6.4. Algorithmic Evaluation

As seen in the algorithmic complexity section 4.3, when the interior builders have similar performance the theoretical speedup can be up to $3\text{-}4\times$, depending on the costs introduced by the primitives reordering, subset sampling and primitives insertion passes, which we will reference as *overhead*. In Figure 8 we can assess how

well the model predicts the final outcome by taking the San Miguel scene as an example. The first two plots compare the performance per level of each building stage: our stochastic builder operates on 20% of the binned SAH data, thus showing a lower amount of processed primitives, until the Stochastic BVH construction is done ((b), red, green and orange). The cluster BVHs build (blue) shows instead a higher count in a short burst due to the final construction phase involving all the primitives. What becomes apparent in the throughput plot is that our assumption over similar performances in BVH construction does not hold in this case. Our stochastic BVH is, in fact, utilizing a bit more than half of the bandwidth while processing a fifth of the data. The cluster BVHs build step instead is showing $1.5\times$ the throughput of the binned BVH. Note how our initial approximation of the final BVH creates a few more levels. This brings down our initial projections to a $2.71\times$ speedup over the construction time alone ((d), *weighted average* bar), and consequently influence the final outcome to a $1.89\times$ ((d), *overhead* bar). By setting α to 0.19 (our measured relative overhead), the expected gain given by our model evaluates at $2.1\times$. The reason behind this behaviour resides in the ability of the chosen builder to saturate the GPU: we can see its effect in (c), where the binned SAH dominates the first half of the construction, only to switch places when the cluster BVHs phase takes place. This highlights how important it is to increase the efficiency of the Stochastic BVH construction and the direct speedup gain that can be obtained, without counting the obvious benefit of a lower overhead.

6.5. Comparison with a Deterministic Clustering

Another approach to reducing top-down BVH build cost is through the use of deterministic clustering of primitives, where a BVH is built inside each cluster, and a top-level BVH is built over the clusters to complete the tree, thus reducing the cost of building the upper levels. Given the simplicity of this clustering-type approach, we seek to compare our stochastic build method to such techniques. For this purpose, we take inspiration from how HLBVH[PL10] forms clusters for constructing an upper level HLBVH hierarchy, but adapt it for our purposes. First, we group all primitives inside the same Morton-coded cell as a cluster BVH, and consider these

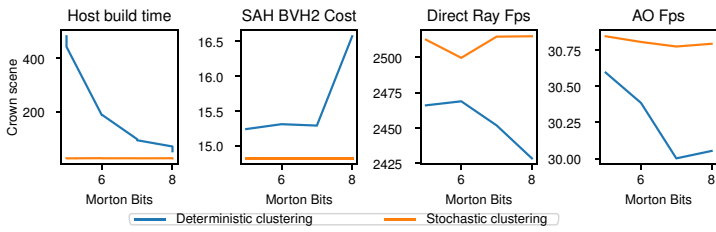


Figure 10: Comparison between different Morton bit count for deterministic clustering and the proposed stochastic approach: low amount of bits results in few clusters fitting the whole geometry, thus paying a huge cost in the bottom phase. High amount instead degenerate in higher SAH cost and overall lower performance.

clusters as primitives to build the top of the tree. Intuitively, LBVH-like approaches suffer from the fixed cell size when scenarios like *teapot in a stadium* arise. This lack of adaptability can be only partially mitigated by our method (see Figure 10): with a low number of bits per axis, the top part of the tree is very small and the benefit is paid by the bottom construction, that now has to build in parallel huge fat leaves. On the other hand, with a higher amount of bits, the method shows the drawback of its rigid structure, with high SAH cost and lower performance. Given these results, an adaptive and ad-hoc construction would be needed to make this approach viable, ending outside our current scope. Nevertheless, we leave it as an interesting venue for future work. For a more extensive analysis we refer to the supplemental.

7. Discussion

Use Cases and Limitations As seen in the algorithmic analysis (section 4.3), our method’s efficiency heavily depends on the ratio between the added overhead and the time taken by the interior builder. Our experiments indicate that the scaling efficiency increases with the primitive’s count, so we recommend using it with large scenes. This limitation is due to the number of kernel dispatches that both our approach and the top-down builder we used require, weighting in the total runtime. The configuration of 20% data for the subset and 10% uniformity for the clamping has been a reliable default in our test, as our technique transparently adapts to different geometry sizes and distributions. Being a stochastic selection, we can incur in under-sampling: small isolated geometry might be missed, resulting in larger bounding boxes in the final BVH. This is partially avoided by the use of stratification and uniformity, which increase sampling guarantees of sparse data.

Relation to Binning Binning [Wal07] distinguishes itself from our approach in that reducing the bin count only accelerates computation of the split, but binning and partitioning still require access to all primitives. Our approach accelerates all parts of construction in the first levels, since the primitive set is smaller. Nevertheless, both approaches are orthogonal and their combination retains their respective strengths, which is why we use a binned builder as the interior builder.

Applicability to other Builders Our algorithm is directly applicable to top-down construction methods like the binned builder we used, but also sweep SAH. Since the computational overhead per level is roughly the same ($O(n \log n)$), the use of subsets in the first levels gives a noticeable performance improvement. Other build algorithms like PLOC which are closer to $O(n)$ do not possess this property. Due to the decreasing set size in each iteration, the top of the tree is already fast to compute. Thus applying our approach to PLOC would give a negligible return if any at all.

Comparison with other Top-Down Build Algorithms In this paper, we introduce the concept of stochastic subsets to the BVH construction problem, and focus on GPU implementation of a top-down BVH builder incorporating this idea. Alternative methods of accelerating top-down construction have previously been demonstrated in the literature [GBDA15; GD16; HMB17]. However, all of these methods are CPU-based, and to our knowledge, none of these methods have been demonstrated on the GPU to date. It would be interesting to compare our GPU-based stochastic method to these builders if an efficient GPU implementation of these algorithms is demonstrated in the future.

8. Conclusion

We have presented a novel approach to BVH construction that leverages the field of sampling, opening new exciting possibilities for cross-pollination. Furthermore, our method transparently selects a subset of the input geometry to improve top-down BVH build performance up to 1.8 \times , while retaining BVH quality competitive with more expensive methods in most cases. Based on our results, we believe that Monte Carlo and Quasi-Monte Carlo techniques introduce a fresh perspective to the problem of fast and high-quality BVH construction.

We see compelling prospects in a number of directions. For example, future work could see the integration of refitting techniques into our subset BVH construction method, or shaping a better constructor that can take full advantage of the subset without loss of performance and reach the full theoretical gain. In addition, using weights and advanced sampling can lead to new optimizations in tree construction that were not possible before. Regarding random sequences, blue noise or random sequences with different properties could also represent an interesting future direction.

Acknowledgments

Model courtesy: Sponza (Crytek), Crown (Martin Lubich), San-Miguel (Guillermo M. Leal Llaguno), Bistro (Amazon Lumberyard), Rungholt (kescha), Hairball (NVIDIA Research). We also want to thank Sebastian Herholz for his feedback on an early draft of this paper, and Anton Kaplanyan and Chuck Lingle for their support on this project.

References

- [AKL13] AILA, TIMO, KARRAS, TERO, and LAINE, SAMULI. “On Quality Metrics of Bounding Volume Hierarchies”. *Proceedings of the 5th High-Performance Graphics Conference*. HPG ’13. Anaheim, California: Association for Computing Machinery, 2013, 101–107. ISBN: 9781450321358. DOI: [10.1145/2492045.2492056](https://doi.org/10.1145/2492045.2492056).

- [AP16] ART B., OWEN and PETER W., GLYNN. *Monte Carlo and Quasi-Monte Carlo Methods*. 1st. Springer Proceedings in Mathematics & Statistics. Springer, 2016. DOI: https://doi.org/10.1007/978-3-319-91436-7_2.
- [BDTD22] BENTHIN, CARSTEN, DRABINSKI, RADOSLAW, TESSARI, LORENZO, and DITTEBRANDT, ADDIS. "PLOC++: Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction Revisited". *Proc. ACM Comput. Graph. Interact. Tech.* 5.3 (July 2022). DOI: [10.1145/3543867_2](https://doi.org/10.1145/3543867_2), 8, 9.
- [BHH13] BITTNER, JIŘI, HAPALA, MICHAL, and HAVRAN, VLASTIMIL. "Fast insertion-based optimization of bounding volume hierarchies". *Computer Graphics Forum*. Vol. 32. Wiley Online Library, 2013, 85–100 2, 6.
- [BHH15] BITTNER, JIŘI, HAPALA, MICHAL, and HAVRAN, VLASTIMIL. "Incremental BVH construction for ray tracing". *Computers & Graphics* 47 (2015), 135–144 2.
- [BM15] BITTNER, JIRÍ and MEISTER, DANIEL. "T-SAH: Animation Optimized Bounding Volume Hierarchies". *Computer Graphics Forum* (2015). DOI: [10.1111/cgf.12581_2](https://doi.org/10.1111/cgf.12581_2).
- [Cor22] CORPORATION, INTEL. *Intel Arc A-Series Graphics*. 2022. URL: <https://ark.intel.com/content/www/us/en/ark/products/series/227957/intel-arc-a-series-graphics.html> 8.
- [DP15] DOMINGUES, LEONARDO R. and PEDRINI, HELIO. "Bounding Volume Hierarchy Optimization through Agglomerative Treelet Restructuring". *High-Performance Graphics*. Ed. by CLARBERG, PETRIK and EISEMANN, ELMAR. ACM Siggraph, 2015. ISBN: 978-1-4503-3707-6. DOI: [10.1145/2790060.2790065_2](https://doi.org/10.1145/2790060.2790065_2), 8.
- [GBDA15] GANESTAM, P., BARRINGER, R., DOGGETT, M., and AKENINE-MÖLLER, T. "Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees". *Journal of Computer Graphics Techniques (JCGT)* 4.3 (Sept. 2015), 23–42. ISSN: 2331-7418. URL: http://jcgt.org/published/0004/03/02/2_12.
- [GD16] GANESTAM, PER and DOGGETT, MICHAEL. "SAH guided spatial split partitioning for fast BVH construction". *Computer Graphics Forum* 35.2 (2016), 285–293. DOI: https://doi.org/10.1111/cgf.12831_2, 12.
- [GHFB13] GU, YAN, HE, YONG, FATAHALIAN, KAYVON, and BLELOCH, GUY. "Efficient BVH Construction via Approximate Agglomerative Clustering". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, 81–88. ISBN: 9781450321358. DOI: [10.1145/2492045.2492054_2](https://doi.org/10.1145/2492045.2492054_2).
- [GPM11] GARANZHA, KIRILL, PANTALEONI, JACOPO, and MCALLISTER, DAVID. "Simpler and Faster HLBVH with Work Queues". *Proc. ACM Comput. Graph. Interact. Tech.* HPG '11 (2011), 59–64. DOI: [10.1145/2018323.2018333_2](https://doi.org/10.1145/2018323.2018333_2).
- [GS87] GOLDSMITH, JEFFREY and SALMON, JOHN. "Automatic creation of object hierarchies for ray tracing". *IEEE Computer Graphics and Applications* 7.5 (1987), 14–20 2, 3.
- [Hes95] HESTERBERG, TIM. "Weighted Average Importance Sampling and Defensive Mixture Distributions". *Technometrics* 37.2 (1995), 185–194. ISSN: 00401706. URL: http://www.jstor.org/stable/1269620_5.
- [HMB17] HENDRICH, JAKUB, MEISTER, DANIEL, and BITTNER, JIRI. "Parallel BVH construction using progressive hierarchical refinement". *Computer Graphics Forum*. Vol. 36. Wiley Online Library, 2017, 487–494 2, 12.
- [KA13] KARRAS, TERO and AILA, TIMO. "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies". *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, 89–99. ISBN: 9781450321358. DOI: [10.1145/2492045.2492055_2](https://doi.org/10.1145/2492045.2492055_2).
- [Kar12] KARRAS, TERO. "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees". *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. EGGH-HPG'12. Paris, France: Eurographics Association, 2012, 33–37. ISBN: 9783905674415 2, 9.
- [KGA*19] KELLER, ALEXANDER, GEORGIEV, ILIYAN, AHMED, ABDALLA, et al. "My Favorite Samples". *ACM SIGGRAPH 2019 Courses*. SIGGRAPH '19. Los Angeles, California: Association for Computing Machinery, 2019. ISBN: 9781450363075. DOI: [10.1145/3305366.3329901_2](https://doi.org/10.1145/3305366.3329901_2).
- [LGS*09] LAUTERBACH, C., GARLAND, M., SENGUPTA, S., et al. "Fast BVH Construction on GPUs". *Computer Graphics Forum* 28.2 (2009), 375–384. DOI: https://doi.org/10.1111/j.1467-8659.2009.01377.x_2_4.
- [LYMT06] LAUTERBACH, CHRISTIAN, YOON, SUNG-EUI, MANOCHA, DINESH, and TUFT, DAVID. "RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs". *2006 IEEE Symposium on Interactive Ray Tracing*. 2006, 39–46. DOI: [10.1109/RT.2006.2802131_8](https://doi.org/10.1109/RT.2006.2802131_8).
- [MB16] MEISTER, DANIEL and BITTNER, JIŘI. "Parallel BVH construction using k-means clustering". *The Visual Computer* 32.6 (2016), 977–987 2.
- [MB18a] MEISTER, D. and BITTNER, J. "Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction". *IEEE Trans. Vis. Comput. Graph.* 24.3 (2018), 1345–1353. DOI: [10.1109/TVCG.2017.2669983_2_6](https://doi.org/10.1109/TVCG.2017.2669983_2_6).
- [MB18b] MEISTER, D. and BITTNER, J. "Parallel Reinsertion for Bounding Volume Hierarchy Optimization". *Computer Graphics Forum* 37.2 (2018), 463–473. DOI: https://doi.org/10.1111/cgf.13376_2.
- [MB22] MEISTER, DANIEL and BITTNER, JIŘI. "Performance Comparison of Bounding Volume Hierarchies for GPU Ray Tracing". *Journal of Computer Graphics Techniques (JCGT)* 11.4 (Oct. 2022), 1–19. ISSN: 2331-7418. URL: <http://jcgt.org/published/0011/04/01/9>.
- [MB90] MACDONALD, DAVID and BOOTH, KELLOGG. "Heuristics for Ray Tracing Using Space Subdivision". *The Visual Computer* 6.3 (1990), 153–65 3, 6.
- [McG17] MCGUIRE, MORGAN. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data_9.
- [MOB*21] MEISTER, DANIEL, OGAKI, SHINJI, BENTHIN, CARSTEN, et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". *Computer Graphics Forum* 40.2 (2021), 683–712. DOI: https://doi.org/10.1111/cgf.142662_2.
- [NT03] NG, KELVIN and TRIFONOV, BORISLAV. "Automatic bounding volume hierarchy generation using stochastic search methods". *Mini-workshop on stochastic search algorithms*. 2003 2.
- [PJH16] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation (3rd ed.)* 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Oct. 2016, 1266. ISBN: 9780128006450 8.
- [PL10] PANTALEONI, J. and LUEBKE, D. "HLBVH: Hierarchical LVBH Construction for Real-Time Ray Tracing of Dynamic Geometry". *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 2010, 87–95 2, 11.
- [VBH17] VINKLER, MAREK, BITTNER, JIRI, and HAVRAN, VLASTIMIL. "Extended Morton codes for high performance bounding volume hierarchy construction". *Proceedings of high performance graphics*. Association for Computing Machinery, 2017, 1–8 2.
- [Wal07] WALD, INGO. "On fast construction of SAH-based bounding volume hierarchies". *2007 IEEE Symposium on Interactive Ray Tracing*. IEEE, 2007, 33–40 2, 8, 12.
- [WBKP08] WALTER, BRUCE, BALA, KAVITA, KULKARNI, MILIND, and PINGALI, KESHAV. "Fast agglomerative clustering for rendering". *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, 81–86. DOI: [10.1109/RT.2008.4634626_2](https://doi.org/10.1109/RT.2008.4634626_2).