Incremental BVH Construction for Ray Tracing

Jiří Bittner, Michal Hapala, Vlastimil Havran

Abstract

We propose a new method for incremental construction of Bounding Volume Hierarchies (BVH). Despite the wide belief that the incremental construction of BVH is inefficient we show that our method incrementally constructs a BVH with quality comparable to the best SAH builders. We illustrate the versatility of the proposed method using a flexible parallelization scheme that opens new possibilities for combining different BVH construction heuristics. We demonstrate the usage of the method in a proof-of-concept application for real-time preview of data streamed over the network. We believe that our method will renew the interest in incremental BVH construction and it will find its applications in ray tracing based remote visualizations and fast previews or in interactive scene editing applications handling very large data sets.

Keywords:
bounding volume hierarchies, ray tracing

1. Introduction

Interactive ray tracing becomes an increasingly popular alternative to rasterization mainly because ray tracing based algorithms allow computing accurate global illumination and thus achieving high degree of realism. One of the main obstacles for their interactive usage is the necessity to organize the scene in an acceleration data structure in order to efficiently compute the ray-object intersection queries. The most commonly used methods involve spatial subdivisions (uniform grids, octrees, kd-trees) and bounding volume hierarchies (BVH). In particular BVHs became a vivid choice for many recent implementations as they have predictable memory footprint, allow relatively easy dynamic updates, and perform well in GPU ray tracing implementations [1].

Practically all currently used BVH build methods require that the whole scene is known in advance. While this is often the case, there are also applications, in which accessing the scene data takes significant amount of time. Waiting for all the data to be present in memory introduces significant latency in the whole rendering process. Another use case when the whole scene is not known in advance is for example an interactive modeling session of complex data assemblies for which high quality preview is required. A natural solution in these applications could be an incremental BVH construction, which inserts pieces of the scene geometry into the BVH as soon as they become available. It is however widely believed that the incremental BVH construction is inefficient particularly in terms of ray tracing performance of the resulting BVH. In this paper we show that using a careful optimization of the incremental BVH construction combined with global structural updates leads to efficient BVHs. In particular we aim at three main contributions: (1) We present an incremental construction algorithm, which produces high quality BVH. We are the first to show that the insertion based incremental BVH construction can lead to efficient BVHs, which directly contradicts the state of the art results [2, 3]. (2) We propose two parallelization schemes of the incremental BVH construction, which are actually the first parallel schemes of incremental BVH construction we are aware of. (3) We test the proposed method in a proof-of-concept application which performs GPU ray tracing of the data streamed over the network while using different data prioritization schemes. An illustration of the incremental BVH construction combined with data streaming is shown in Figure 1.

2. Related Work

Bounding volume hierarchies provide an efficient way of organizing scene primitives and they have a long tradition in the context of ray tracing. Already in the early 80s Rubin and Whitted [4] used a manually created BVH, while Weghorst et al. [5] proposed to build the BVH using the modeling hierarchy. Kay and Kajiya [6] designed a top down BVH construction algorithm using spatial median splits. Goldsmith and Salmon [7] proposed the measure currently known as the *surface area heuristic* (SAH) which predicts the efficiency of the hierarchy already during the BVH construction. In this highly influential work Goldsmith and Salmon proposed to build BVH incrementally by insertion. However the algorithm they provided was limited to greedy decisions during the insertion process and did not properly explore the space of all possible insertion positions. This insertion based method thus generally results in a poor quality BVH as was shown in performance studies by Havran [2] and later by Masso et al. [3]. This led to a belief that the incremental construction of a BVH by insertion is inefficient and these methods were practically disregarded by the research community. In our paper we revisit the idea of incremental BVH construction and show that it can actually lead to trees of higher quality than the nowadays used top-down SAH construction methods.

0.1% (13k tris) | 0.5% (63k tris) | 1% (127k tris) | 2% (255k tris) | 100% (12.7M tris)

Figure 1: Snapshots showing ray traced images of the Power Plant scene (12.7M triangles) during data streaming. A high quality BVH is constructed incrementally on the CPU, while the scene is being ray traced on the GPU at real-time (60FPS). The data is sent by prioritizing the geometry based on its estimated projected area. By streaming a fraction of the scene geometry we already obtain a good overview of the visible part of the scene.

Bounding volume hierarchy construction was also studied in the context of collision detection, for which Omohundro [8] designed an efficient method using a priority queue based search for construction of a hierarchy of bounding spheres. A similar search strategy was recently used by Bittner et al. [9] in an algorithm, which optimizes the BVH in a postprocess. This work however gives no indication if the proposed optimization methods can also be used for the actual construction of high quality BVHs.

The vast majority of currently used methods for BVH construction use a top-down approach together with the surface area heuristic [10]. These methods require sorting and thus generally exhibit $O(NlogN)$ complexity (N is the number of scene triangles). Several techniques have been proposed to reduce the constants behind the asymptotic complexity. For example Havran et al. [11], Wald et al. [10], and Ize et al. [12] used approximate SAH cost function evaluation based on binning. Hunt et al. [13] suggested to use the structure of the scene graph to speed up the BVH construction process. Dammertz et al. [14] proposed to use a higher branching factor of the BVH to better exploit SIMD units in modern CPUs. More recently, the parallel build-up of a BVH has been demonstrated also on a GPU by Lauterbach et al. [15], using a 3D space-filling curve. Aila and Laine [1] targeted optimization of BVH traversal on the GPU. Wald studied the possibility of fast rebuilds from scratch on an upcoming Intel architecture with many cores [16]. Pantaleoni and Luebke [17], Garanzha et al. [18], and Karras [19] proposed GPU based methods for parallel BVH construction. These methods achieve impressive performance, but generally construct a BVH of lower quality than the full SAH builders.

Recently more interest has been devoted to methods, which are not limited to the top-down BVH construction. Walter et al. [20] propose to use bottom-up agglomerative clustering for constructing a high quality BVH. Gu et al. [21] propose a parallel approximative agglomerative clustering for accelerating the bottom BVH construction. Kensler [22], Bittner et al. [9], and Karras and Aila [23] propose to optimize the BVH by performing topological modifications of the existing tree. These approaches allow to decrease the expected cost of a BVH beyond the cost achieved by the traditional top down approach. The comparison of different BVH construction methods and new quality metrics have been studied recently by Aila et al. [24].

Our paper makes use of the incremental BVH construction in an application, which receives streamed scene data over the network. This area has been thoroughly researched particularly in the case of massive model visualizations [25, 26]. These methods typically use specialized scene representations (such as LODs, point clouds, or voxels) and work usually with the rasterization paradigm rather than ray tracing. In our paper the streaming component is used only as a particular use case of the proposed incremental BVH construction and thus for more details about the remote and out-of-core visualization techniques we direct an interested reader to the survey of Gobetti et al. [27].

The paper is further structured as follows: The overview of the algorithm is given in Section 3. The incremental BVH construction algorithm is described in Section 4 and its parallelization in Section 5. Section 6 presents the framework, which exploits the proposed BVH construction for ray tracing data streamed over the network. Section 7 presents the results which are discussed in Section 8. Finally, Section 9 concludes the paper.

3. Algorithm Overview

The core of our method is the incremental insertion of scene geometry into the BVH. In the sequential version of the algorithm we construct a new leaf node for each geometric primitive (triangle), which is then inserted at an appropriate position into the BVH. We use a branch and bound search to find a position in the tree which minimizes the increase of the tree cost evaluated using SAH. The new leaf is then linked to the tree and the process continues with the next geometric primitive. Apart from the sequential algorithm we propose two methods of parallelization of the algorithm. The first method searches for the best positions of the triangles in the BVH for a batch of triangles in parallel. The second method subdivides the input triangle stream into chunks for which small local BVHs are constructed in parallel and then sequentially inserted into the global BVH.

The final BVH quality depends on the order of inserted primitives - for some orders the tree might get globally imbalanced with respect to the SAH cost metric. We compensate for that by performing global tree updates by re-inserting selected nodes at better positions in the BVH so the global BVH cost is minimized. The selection of nodes for re-insertion is driven by tracking the history of BVH modifications performed for the inserted geometry.

The BVH construction can handle input geometry provided in arbitrary order. We also discuss view dependent prioritization schemes which change the order in which the data is streamed. These methods are based on evaluating the importance of scene primitives for the current camera view and using either a deterministic or a stochastic approach for prioritizing the data according to their importance.

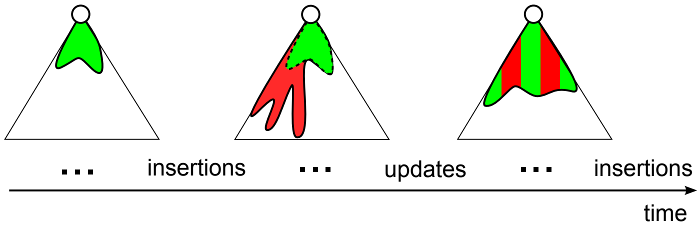


Figure 2: Illustration of the interleaving of insertion and update operations. The incremental insertion of nodes is searching for the best position of inserted nodes, however the overall structure of the tree might get imbalanced. This is corrected by BVH updates, which aim to globally optimize the current tree. Note that unlike this illustration, the tree optimized according to SAH will typically not be balanced in terms of depths.

4. Incremental BVH Construction

In this section we recall the SAH cost model and then we present the incremental BVH construction, which forms a core contribution of our paper.

4.1. SAH Cost Model

The quality of the BVH for ray tracing purposes is commonly measured using the SAH cost model, which expresses the expected number of operations to process a ray intersecting the scene. This cost can be expressed as:

$$C(T) = \frac{1}{S(T)}\left[c_T \cdot \sum_{N \in inner\ nodes} S(N) + c_I \cdot \sum_{N \in leaves} S(N) \cdot t_N\right], \quad (1)$$

where $S(T)$ is the surface area of the bounding box of the scene, $S(N)$ is the surface area of the bounding box of node N, t_N is the number of triangles in leaf N, and c_T and c_I are constants representing the traversal and intersection costs. Note that the cost of intersecting the triangles in the leaves is constant for a given scene supposed there is a single primitive per leaf. Thus the cost term which should be minimized when inserting new primitives is the sum of surface areas of inner nodes in the tree which corresponds to the traversal overhead of the interior part of the tree ($c_T \cdot \sum S(N)$).

4.2. Inserting Primitives

The geometric primitives are inserted into the BVH incrementally, one by one. For each primitive we first create a new leaf node containing this primitive. Then we need to find an appropriate position in the BVH where the node should be inserted. For this purpose we use the branch and bound algorithm proposed by Bittner et al. [9], which was originally designed for BVH optimization by repositioning its subtrees. This algorithm searches for a node in the tree which will become the sibling of the inserted node, such that the global cost increase given by Eq. 1 is minimized.

4.3. Global BVH Updates

The primitive insertion step of the algorithm finds an optimal position of the node with respect to the current BVH topology, but without reflecting primitives that will be inserted later. Therefore, in general, the tree might get imbalanced with respect to the SAH metric, since the order of insertions is also important. We solve this problem by interleaving the primitive insertion with a small number of global updates of the BVH. In particular we perform a batch of insertion operations followed by a batch of tree update operations. The process of interleaving insertions and updates is illustrated in Figure 2.

The global updates work by selecting a number of nodes whose children are removed from the tree and then reinserted at better positions in the tree. The nodes are selected using a metric which aims to identify those nodes that cause a cost overhead and thus the re-insertion procedure applied on these nodes has a higher chance for reducing the tree cost. Bittner et al. [9] proposed to use a combined inefficiency measure. We observed that this measure also works well for the optimization during incremental BVH construction. As an alternative approach we can use the surface area of the node as its inefficiency measure, which gives only marginally worse results.

Node update cache. During the incremental construction it is often the case that only some branches of the tree are modified by subsequent insertion operations. We exploit this observation by keeping a cache of nodes for which their bounding box has been modified by insertion in a given batch of insertion operations. These nodes correspond to the union of paths in the BVH from the inserted leaves towards the root (see Figure 3). The update procedure then uses only the cached nodes when selecting the nodes to be updated. We use two constants in our algorithm: the first constant N_u gives the number of modified nodes, reaching which the batch of update operations is applied. The second constant k_u is a fraction of nodes to be updated in a batch: $k_u.N_u$ nodes with the highest inefficiency metric are updated in the given batch. Setting larger N_u increases the size of the length of the insertion batch, while the length of the update batch is given by both constants. We used $N_u = 8000$ and $k_u = 1\%$, which works well for the tested scenes. We observed that the proposed algorithm is generally not very sensitive to these two constants.

4.4. Optimizations

Clustering subsequent primitives. Although the algorithm stated above assumes no particular order of scene primitives, it is often the case that these are already ordered in a spatially coherent way. We can use a simple optimization which makes use of such coherence to reduce the number of insertion operations. In particular we check whether two consecutively inserted primitives are spatially coherent and if this is the case we connect the leaves representing these primitives to form a small

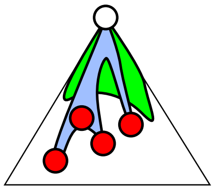


Figure 3: Selecting candidate nodes for topological updates. Several new leaves were added to the tree (shown in red). The part of the tree for which the bounding boxes have been modified corresponds to the candidate nodes for the update (shown in blue). Note the unmodified part of the tree which does not serve for candidate selection (shown in green).

subtree with a single inner node. Then this subtree is inserted into the BVH using a single insertion operation. The coherence of two primitives x, y is measured using the ratio of the surface area of the union of their bounding boxes and the sum of surface areas of the bounding boxes:

$$R_{coh}(x, y) = \frac{S(x \cup y)}{S(x) + S(y)}$$

If $R_{coh}(x, y) < R_{max}$ (we used $R_{max} = 1.5$), the primitives are assumed to be coherent and they are connected to form a subtree which is inserted into the BVH as a whole. This simple optimization brings up to 30% speedup for some scenes, while reaching a very similar BVH cost.

BVH postprocessing. Another possible optimization is to perform a larger batch of update operations after the incremental BVH construction has been finalized [9]. Note that we did not use this optimization in order to present the raw results for the incremental BVH construction for the streamed triangle data.

5. Parallel Incremental BVH Construction

The incremental BVH construction processing individual triangles is inherently sequential, i.e. the BVH is constructed by subsequently extending the current BVH one triangle at a time. The amount of parallelism exploitable while inserting a single triangle into the BVH is limited, since the branch-and-bound search procedure performs localized search and thus does not visit too many nodes of the tree.

However if we subdivide the input stream into batches of triangles of a given size, we can exploit parallelism while inserting the triangle batch into the BVH. We propose two conceptually different ways of parallelizing the incremental BVH construction, parallel search and block parallel construction. Later in the results section we will show that the choice of the method depends on the properties of the input triangle stream and also on the desired BVH quality. Note that both methods have been designed to exploit multi core CPUs rather than GPUs. This matches our target application that will be described in Section 6, in which we aim to fully utilize the GPU for rendering in order to maximize ray tracing performance.

5.1. Parallel Search

The most costly operation in the BVH construction is the search for the position of the currently inserted node in the tree. Thus by parallelizing this operation we can speed up the whole BVH construction process. We execute the branch-and-bound search algorithm in a number of threads for all nodes corresponding to the triangles in the batch. As a result of this parallel operation each node is assigned a node in the BVH to be connected with. Then the nodes are inserted into the BVH sequentially. Using sequential linking into the tree prevents conflicts of threads inserting a node into the same position in the tree. The algorithm based on parallel search is illustrated in Figure 4.

For implementing the method we have used Intel's Thread Building Blocks (TBB) library, which is extremely simple to use and also handles efficient scheduling of the threads. Note that it is beneficial to use a small batch size roughly corresponding to the number of threads used for the search. Larger batch sizes decrease the quality of the constructed BVH as the results of the search do not reflect the positions of the triangles from the same batch.

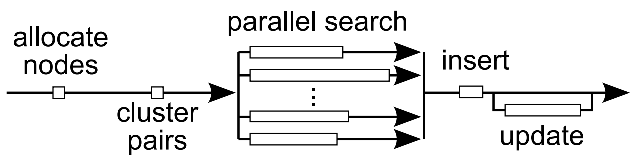


Figure 4: Illustration of parallelization of the search phase of the BVH construction algorithm. Note that the length of the white rectangles roughly corresponds to the costs of the individual steps of the algorithm.

5.2. Block Parallel Construction

The parallelization scheme described above does not provide a linear speedup. This is mainly due to the sequential insertion phase and the associated need of synchronizing the search threads. We can improve the scalability of the algorithm by using a different parallelization scheme in which the CPU cores will get better utilized.

The idea of this parallelization scheme is to create a number of larger triangle batches for which we invoke parallel construction of small BVHs representing the triangles in the batch. We denote these small trees *bBVH* (batch BVH). The bBVHs are fed to a thread which inserts them into the global BVH. In both cases we use the insertion based method. Note that in the case of the bBVHs they can be constructed by any existing method since all triangles in the batch are known when the construction of the bBVH is invoked. Apart from the input triangle buffer the method uses two work queues: the first queue contains the batches for which bBVHs should be constructed. The second queue contains the already constructed bBVHs which should be inserted into the global BVH. The overview of this parallelization method is shown in Figure 5.

If the input triangle stream is coherent, we can create batches of triangles just by grouping the consecutive triangles in the input stream. However for incoherent streams such method would lead to a low quality BVH as the bBVHs might contain incoherent geometry and in turn the bBVHs would have significant spatial overlaps. We handle this issue by creating the triangle

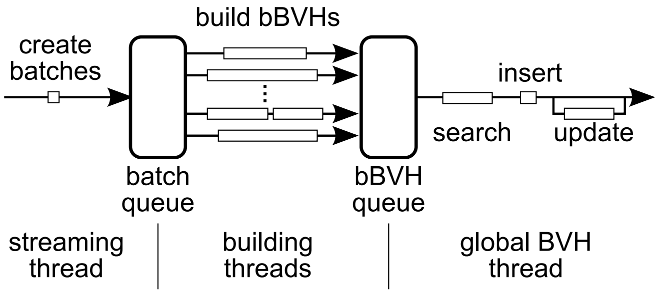


Figure 5: Illustration of the block parallel BVH construction algorithm. Streaming thread creates coherent triangle batches, building threads construct bBVHs for the batches in parallel, and the global BVH thread inserts the constructed bBVHs into the global BVH.

batches by spatial sorting the buffered input stream. The triangles currently available in the buffer are sorted using a quick-sort like approach corresponding to spatial median splits.

Initially all currently buffered triangles form one batch. We evaluate whether the triangles in the batch B are sufficiently coherent using an extension of the above defined coherency measure:

$$R^*_{coh}(B) = \frac{S(B)}{\sum_{i \in B} S(i)} \sqrt[3]{|B| - 1},$$

where $S(B)$ is the surface area of the bounding box of the triangle batch, $|B|$ is the number of triangles in the batch, $S(i)$ is the surface area of the bounding box of the triangle i. Note that the extension is derived so that for two triangles $R^*_{coh}(\{x, y\}) = R_{coh}(x, y)$ and for larger batches $R^*_{coh}(B) \approx 1$ if the bounding boxes of the triangles form cells of a regular 3D grid.

If $R^*_{coh}(B)$ is smaller than a threshold R_{max}, we consider the batch to be coherent and send it for processing without further subdivision. Otherwise, if $R^*_{coh}(B) \geq R_{max}$, the batch is incoherent and needs to be subdivided. We use a cycling axis spatial median pivot (center of the bounding box of the batch in the current axis) to sort the triangles into two groups according to the pivot. This process repeats until the coherency criterion is met or we have a single triangle in the batch.

6. Ray Tracing Streamed Data

Our method is capable of adding new primitives to an already built BVH without reducing its quality and therefore its possible application lies for example in rendering scenes that are received in parts. This may involve either very large data sets, for which it is impractical to wait until the storage medium provides the whole set, or data streamed over a network, where it may take a long time untill the next part arrives. In these cases our method can provide an interactive ray traced visualization of the data set even when it is not complete.

6.1. Application Architecture

We designed and implemented a pilot application, which is capable of real-time ray tracing of data streamed over a network. The application contains client and server parts. For each connected client the server provides the client the objects representing the scene data using a certain data prioritization scheme. The client application inserts all received objects into the BVH using the proposed incremental algorithm and renders them using the GPU based ray-tracer by Karras et al. [28]. The client also informs the server of any camera changes, since this is necessary for the computation of some of the prioritization metrics. The overview of the application framework is shown in Figure 6.

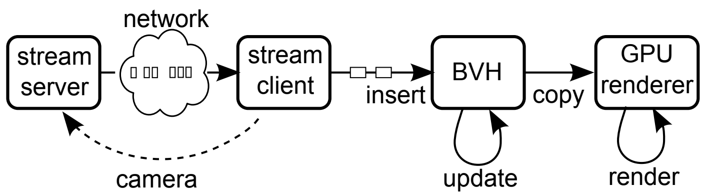


Figure 6: Overview of the application framework for ray tracing streaming data with the incremental BVH construction at its core.

6.2. Data Prioritization

In the early stages of the rendering session the visualized scene data are incomplete. In order to evaluate our incremental construction we used different prioritization schemes for the streamed data. In particular we have tested the following four prioritization schemes:

The *view direction* prioritization scheme uses a dot product of the view direction and the vector from the camera position towards the object (triangle) as the priority of the object. We used a deterministic algorithm, which at each step selects a batch of k untrasferred objects with highest priorities using a partial sort.

The *projected area* prioritization uses the estimated projected area of the object as its priority. For this scheme we used stochastic sampling algorithm that constructs a cumulative distribution function (CDF) and uses it to randomly draw the objects to be sent with probability proportional to the priorities. To select an object to be sent we generate a uniformly distributed random number which is mapped to a particular object index by using a binary search in the CDF.

The *as is* scheme involves no prioritization and is suitable for the case when the camera parameters are not available at the server side or when the server could get overloaded by evaluating the view dependent client prioritization schemes.

The *random* scheme sends the scene objects in a random order. This allows to test how the incremental construction handles incoherent data both in terms of speed and BVH quality.

7. Results

We have implemented the proposed incremental BVH construction method in C++. The GPU ray tracing part is implemented using CUDA. The results were evaluated on a PC with Intel Xeon E5-1620/3.60GHz CPU (4 cores) with 16GBytes RAM, equipped with NVIDIA GeForce GTX 580 GPU with 3GBytes RAM. For measurements we used nine test scenes which are summarized in Figure 7.

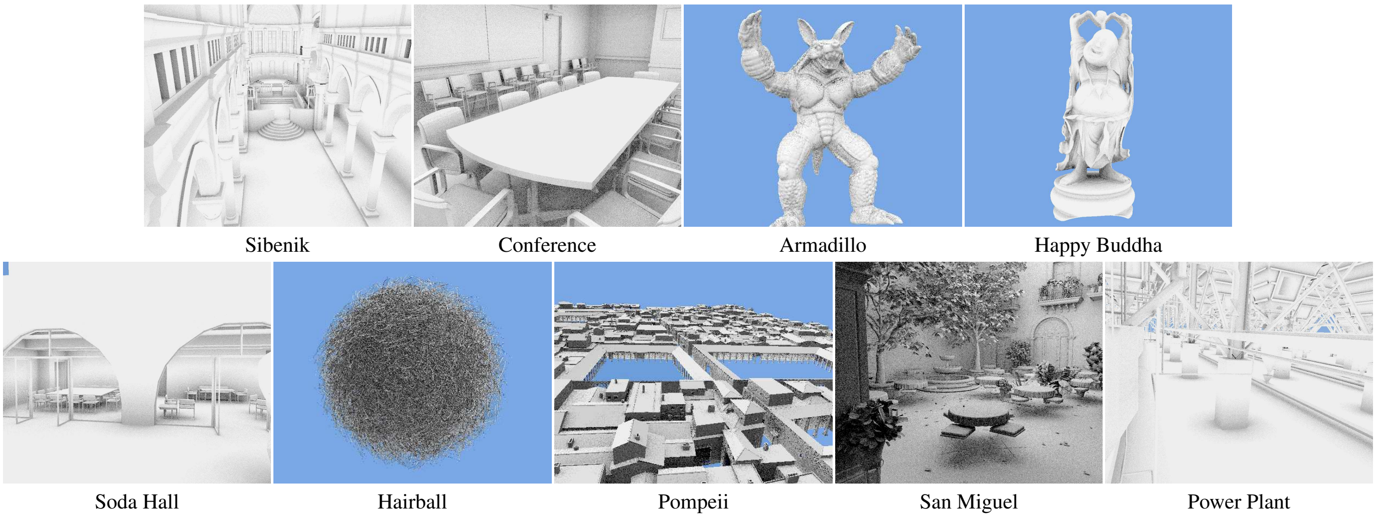


Figure 7: Snapshots of the tested scenes.

7.1. *Incremental BVH Construction*

First we evaluated the proposed incremental BVH construction algorithms. We focused on the construction time and the resulting quality of the BVH. The quality was expressed using the SAH cost of the BVH and also by measuring the GPU ray casting performance. As reference methods we used a BVH constructed by a high quality sweep-based SAH builder (denoted SAH) and by spatial median splits (denoted Median). For our proposed algorithm we evaluated four versions: the first one (Incr) uses only insertion operations and performs no global updates, the second one (IncrU) uses the global updates, the third one (IncrUP) uses parallel search and global updates, and the fourth one (IncrUPB) uses block parallel construction with global updates. The parameters for the global updates were $N_u = 8000$ and $k_u = 1\%$. We have used three different stream ordering methods: as is, view direction prioritization, and random. Note that the random order represents an extreme case for the incremental insertion build as there is almost no coherence among consecutive triangles in the stream. The measured results are summarized in Table 1.

Build time. The results show that even the sequential implementations of the proposed methods (Incr, IncrU) are always significantly faster than the full sweep SAH builder (SAH) in terms of BVH construction speed. For coherent stream orders they are about twice slower than the spatial median algorithm (Median), but this gap gets larger for random ordering. We can also observe that the IncrU method is faster than Incr for all cases except for the random stream order. This is due to the fact that the method continuously works with a slightly more optimized BVH, which also reduces the cost of insertion operations. The parallel search based implementation of the method IncrUP is about $15-50\%$ faster than IncrU, while the block parallel method IncrUPB is up to 5 times faster than IncrU. However for random stream order the speed benefit of the IncrUPB method reduces and it can even get slower than the IncrUP method.

BVH cost. Regarding the quality of the constructed BVH we can observe that in most cases both incremental construction methods construct a BVH with even lower cost than the full top-down SAH builder. In particular the BVH constructed with IncrU method has usually about 10% lower cost than the BVH constructed with full SAH. An exception when the BVH cost for the incremental construction is higher than SAH is the Happy Buddha scene. An interesting observation is that the random stream order leads to higher quality BVH for the incremental methods. This is however paid by significantly longer construction times.

Streaming speed. We also expressed the average streaming throughput for the incremental BVH construction expressed in millions of triangles per second inserted into the BVH (MTris/s). This throughput varies among the tested scenes in the range of 0.1 - 0.8 MTris/s for the sequential implementation and 0.1 - 2.9 MTris/s for parallel implementation. When comparing the speed versus quality of the different incremental construction methods we can observe that the IncrUP would be the method of choice when the BVH quality is important. On the other hand the IncrUPB method is a good choice when maximum streaming throughput is desired.

Ray tracing speed. Table 1 also shows the measured GPU ray tracing performance for the final BVH constructed by the different methods expressed in millions of rays per second (MRays/s) for two different ray types (primary rays and ambient occlusion rays). For all the proposed methods the measured performance varies between 25-294 MRays/s and allows real-time ray tracing of the tested scenes. We can observe that the highest rendering performance is mostly obtained using the IncrU or IncrUP methods, while the block parallel IncrUPB method usually achieves slightly lower ray tracing performance.

Progress of the computation. To evaluate the progress of the incremental BVH construction we show the number of processed triangles as a function of time (Figure 8-left). We observed that the triangle insertion throughput slightly decreases as the BVH contains more nodes, but this dependence is very weak. This conforms with the theoretic logarithmic decay of the triangle insertion throughput. Figure 8-middle shows that the BVH cost has generally non uniform evolution as we can observe also the sudden reductions of the BVH cost in time which are caused by a successful batch of update operations. Note that for the case of random triangle order the cost evolution curve is much smoother (see Figure 8-right). Figure 9 shows a detailed comparison of the BVH cost evolution for different streaming strategies on three selected scenes. To give an idea how frequent the global BVH updates are we measured the relative number of update operations expressed as the number of update operations with respect to the number of triangles in the scene. This value varies among 0.6-1.7%, so a relatively low number of update operations is able to keep the tree well balanced.

We also tested the influence of changing the number of updated nodes per batch (k_u). When increasing k_u from 1% to 5%, we observed a marginal increase of build time in order of 1% to 5% and also a reduction of the BVH cost in order of few percent for vast majority of tests. In some cases the reduction of the BVH cost was even more significant (e.g. 20% lower cost for IncrU on Happy Buddha at 5% increase of build time). However, in some other cases the time increase was higher, but it was not reflected in the higher cost reduction (e.g. 30% increase of build time with 2% cost reduction for IncrU at San Miguel).

7.2. Ray Tracing Streaming Data

In order to evaluate the sample application using network streaming we captured several videos showing the behavior of the application depending on the data prioritization method and network bandwidth (the videos are provided as a supplementary material for the paper). Several snapshots showing the application at different stages of data streaming are shown in Figure 1.

The projected area based prioritization provides a very fast global overview of the scene structure, however due to its inherent stochastic nature the scene contains a lot of noise appearing as cluttered geometry. The view direction prioritization on the other hand quickly reveals the details in the area of camera focus, while it takes longer to give the global scene structure. In our tests we generally found the view direction method more pleasant to use and very intuitive - when the user moves the camera the method automatically streams the part of the scene in the new camera focus.

We also measured the GPU ray tracing performance in dependence on the number of received triangles for the different streaming strategies (see Figure 10). We observed that for the projected area based prioritization the ray tracing speed reduces faster than for the other two methods. This follows from the fact that this prioritization technique is designed to fill the rendered image with objects as fast as possible (most rays intersect some visible objects at early stages of the computation). The other two methods fill the image more gradually, which as a side product is reflected in the slower reduction of the rendering speed. Note that even for the final BVH with several million triangles, the rendering speed is sufficient for interactive ray tracing of the scene as shown in Table 1.

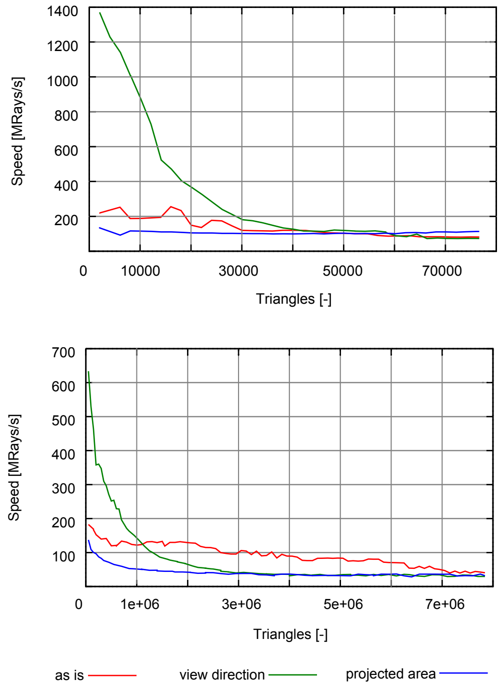


Figure 10: Performance of the GPU ray tracing depending on the number of triangles inserted into the BVH. The graph shows different streaming prioritization methods measured on the Sibenik (top) and San Miguel scene (bottom).

8. Discussion and Limitations

BVH cost. The results show that the proposed method constructs a very high quality BVH for most tested scenes. However we have observed that for some scenes with a simpler and more regular structure the methods performs slightly worse than the top-down SAH (e.g. HappyBuddha, Armadillo). This can be compensated by subsequent update passes applied on such scenes [9].

Comparison to Goldsmith and Salmon. The only previously proposed and evaluated incremental BVH construction method for ray tracing is the technique proposed in the highly influential paper of Goldsmith and Salmon [7]. This paper contains rather vague description of the actual algorithm, however the results obtained by different implementations of the method [2, 3] show that our technique creates more than an order of magnitude better BVH in terms of its cost, particularly for larger scenes for which the method of Goldsmith and Salmon fails to construct a BVH comparable with the top-down SAH builders.

Construction Speed. The proposed methods achieve construction speeds of 0.1-2.9MTris/s. This is on one hand much higher than the equivalent speed of the reference full SAH builder, on the other hand lower than the speed of the fast GPU builders [17, 18]. A benefit of the proposed method is that by performing the construction on the CPU, the GPU can ray trace the scene in real-time without being forced to offload its resources to the BVH construction. Another important benefit is the reduced

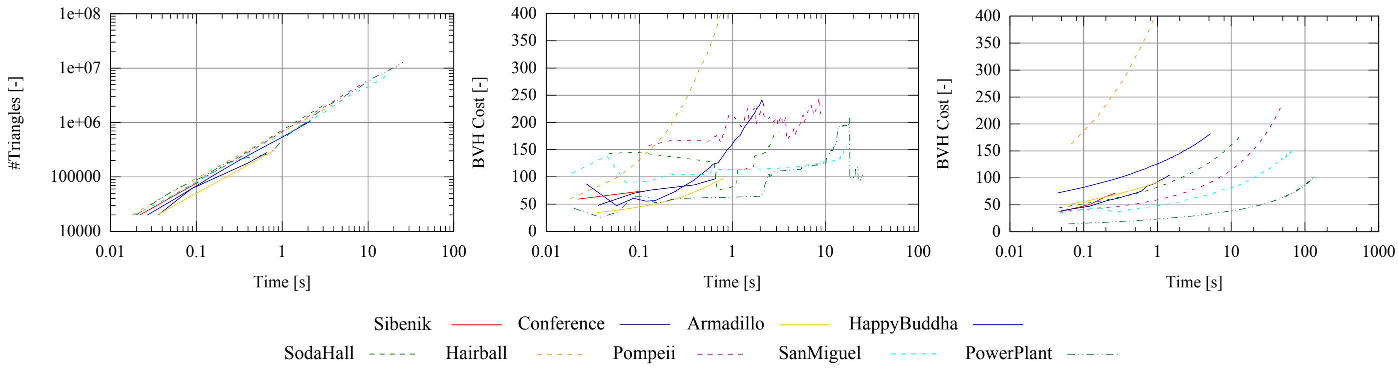


Figure 8: (left) The number of inserted triangles as a function of time for all tested scenes using as is triangle order. (middle) The evolution of the BVH cost during the BVH construction using as is triangle order. We can observe moments when the cost was decreased due to the global BVH updates. (right) The evolution of the BVH cost during the BVH construction using random triangle order. Note the logarithmic scales of the graphs.

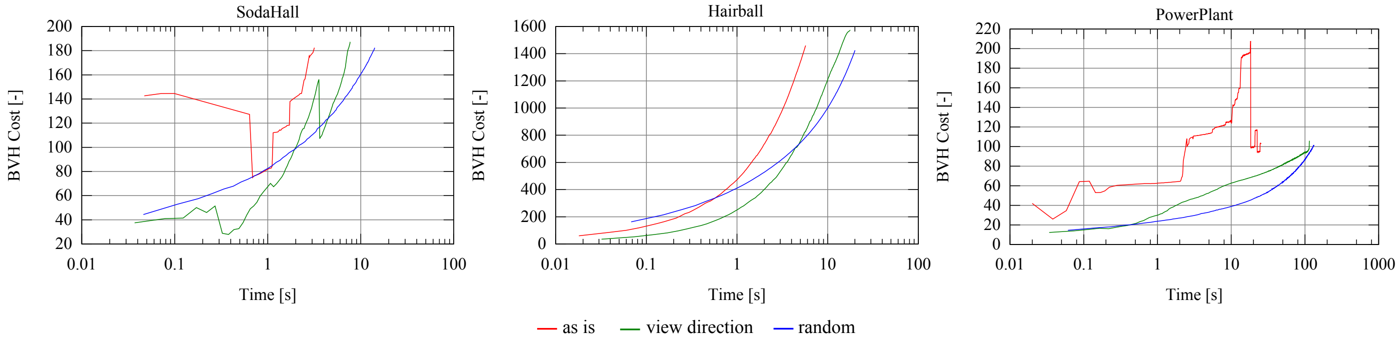


Figure 9: The evolution of the BVH cost during the BVH construction for the IncrU method measured on Soda Hall, Hairball, and Power Plant scenes. We can observe moments when the cost was decreased due to the global BVH updates, especially in the case of *as is* stream order. Note that the random stream order causes smooth BVH cost evolution and leads to slightly lower final BVH cost at the expense of higher computational time.

latency of the rendered image. In particular if the construction speed in MTris/s is higher or comparable to the streaming throughput our method leads to minimal latency in the appearance of the data on the screen regardless of the scene complexity. The latency is caused only by inserting either a single triangle or a batch of triangles into the tree. Note that the latency reduction is useful also for loading large data sets from the disk. It is often the case that the data is stored in a format which needs decompression and parsing and thus the streaming throughput of the parser in MTris/s is similar to the speed of our incremental construction algorithm. That means that as soon as the parsing of the scene is finished, the BVH is already available and can be used for rendering.

Latency Analysis and Comparison. We conducted a comparison, which aims at defining a use case for which the incremental BVH construction outperforms the existing fast CPU and GPU builders. The comparison is based on the recent results reported by Karras and Aila [23] and Gu et al.[21].

For the comparison we use the San Miguel scene with building times and ray traversal performance reported in the original papers. For the method of Gu et al. we scaled the reported building performance to four core CPU to make the results comparable to the ones measured on our hardware. We evaluate the latency of appearance of a batch of triangles once the batch is received by the test application. For the non-incremental meth-

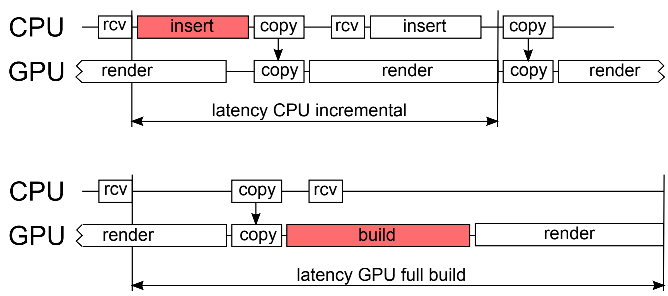
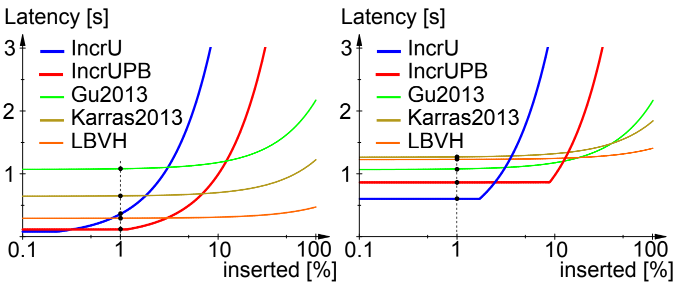


Figure 11: The main components of the latency of appearance of newly received geometry. (top) Latency for CPU incremantal construction. Note that if the newly inserted geometry is small enough the insertion time is completelly hidden by the rendering time and thus the latency is given only by copy and rendering times. (bottom) Latency for full build on the GPU.

ods we assume that the BVH is rebuilt from scratch when the batch of triangles is received. The latency has three main components: time for copying the new data to the GPU, time for building/updating the BVH, and time for rendering the frame (see Figure 11). For the GPU builders (denoted Karras2013 and LBVH) the latency can be approximated as: $t_l = 2(pN_T/s_C + (1+p)N_T/s_B + N_R/s_R)$, where p is the relative number of newly inserted triangles, N_T is the number of scene triangles, s_C is the speed of copying the triangles from CPU to GPU, s_B is the construction speed, N_R is the number of rays cast for one frame, and s_R is the speed of tracing the rays with the given BVH. For the CPU builder proposed by Gu et al. [21] (denoted Gu2013) the latency is expressed as $t_l = 2(pN_T/s_C + max((1+p)N_T/s_B, N_R/s_R))$ since the CPU building and GPU rendering can run in parallel. For the proposed incremental methods (IncrU and IncrUPB) the latency is expressed as $t_l = 2(pN_T/s_C + max(pN_T/s_B, N_R/s_R))$ since the insertion and GPU rendering runs in parallel and furthemore we only insert the new triangles in the tree. Note that in the latency models we assume that the triangle insertion speed and the ray tracing speed are constant for the given method, which does not hold especially when p is large as both are influenced by the newly inserted triangles. However, we target at the use case when p is small for which this approximation is sufficient.



Method	s_B [MTris/s]	s_R [MRays/s]	4M t_l [ms]	30M t_l [ms]
IncrU	0.44	99	359	**605**
IncrUPB	1.60	69	**116**	866
LBVH	107.0	55	294	1081
Karras2013	29.0	84	650	1272
Gu2013	14.8	92	1081	1234

Figure 12: The comparison of rendering latency for different BVH construction methods when inserting a batch of new triangles in the scene. The plots and the table show the latency in dependence on the size of the inserted batch for the SanMiguel scene. (top) Casting 4M rays per frame. (middle) Casting 30M rays per frame. (bottom) The table showing parameters used for compared methods and the evaluated latency for the case of inserting 1% of new scene triangles and tracing either 4M or 30M rays. s_B is the construction speed, and s_R is the ray tracing speed, t_l is the evaluated latency. Note that the CPU to GPU transfer speed was set to $s_C = 500MTris/s$ for all methods.

The results of the comparison for small number of rays per frame (4M) and larger number of rays per frame (30M) are shown in Figure 12. We can observe that with 4M rays per frame the incremental construction (methods IncrU and IncrUPB) lead to significantly lower latency for small values of p. Observe that for the incremental methods the latency is constant for small batches as it is solely given by copy and rendering times. Therefore the benefit of the incremental construction would become even more apparent if lower number of rays would be cast. For larger batches (> 3% of scene size) the slower triangle throughput of the incremental insertion becomes more apparent and the LBVH method leads to the smallest latency among compared methods. For higher number of rays shown in the second plot the situation is similar for small batches of inserted triangles although the latency reduction is not that significant anymore as the tracing time becomes more significant. The incremental methods provide the best results until the batch size of 12% of scene size. For a short interval of batch sizes (12%-17%) the method of Gu et al. provides the best results as it is relatively fast and provides a high quality BVH, while for the even larger batches again the LBVH method leads to the smallest latency. To summarize the latency analysis, we conclude, that our method significantly reduces the latency compared to the state of the art full-build methods for the case of incrementally inserting batches of triangles forming only a fraction of the scene size.

Implementation. The implementation of the method is straightforward and particularly in its sequential version it is much simpler than that of the other high quality BVH builders. This makes the method a good choice for rapid prototyping of applications requiring high quality BVH. In more complex projects the method can coexist with other BVH construction / update implementations (running either on CPU or GPU) and the one most efficient for target application should be used.

Limitations. As the main limitation of the method we see the need for synchronization of the insertion and update operations. The proposed parallelization methods are able to partially remove this limitation. However, the parallel search method does not scale well to larger number of threads. The block parallel construction scales well except for the random triangle order and generally leads to trees of slightly lower quality. The scalability of the method might be improved by a combination of insertion based construction with a different build strategy, but we leave this as a topic for future work. Additional issue which would have to be addressed in the actual streaming based application is handling materials and particularly textures. As textures are typically defined over larger geometric groups the streaming should take texture information into account when determining a geometry order providing the fastest visual feedback.

Data Prioritization. We used three basic strategies for data prioritization in order to demonstrate the possibilities of the proposed incremental BVH construction. There are numerous alternatives how to prioritize the data and also how to incorporate scalable geometric representation by using LOD techniques. A deeper evaluation of the different streaming strategies and associated LOD methods goes out of the scope of our paper, in which the core contribution is the incremental BVH construction algorithm and its evaluation.

9. Conclusion

We have proposed an incremental BVH construction algorithm, which constructs a BVH with better or comparable quality than the traditional SAH based top-down BVH construction methods. The proposed method debunks the myth of insertion based BVH construction not being competitive with the top-down BVH construction. The sequential implementation of the algorithm achieves construction speeds up to 0.8 million triangles per second, and the parallel algorithm achieves speeds up to 2.9 million triangles per second on a 4 core CPU. This makes the proposed method significantly faster compared with the reference implementation of the precise top-down SAH build.

We have shown a possible application of the method for real-time ray tracing of scenes which are streamed over a network. This application uses GPU ray tracing, while the networking layer and the incremental BVH construction is implemented on the CPU. We have used several simple prioritization schemes allowing fast previewing of large data sets even in the case of low network bandwidth. We believe that our method has a prospective use in mobile setups when streaming data over the network. In the future we would like to study other possible applications of the incremental BVH construction such as LOD methods or handling large scale online virtual worlds.

Acknowledgements

We would like to thank Marko Dabrovic for the Sibenik model, Greg Ward for the Conference model, Carlo H. Séquin for the Sodahall model, Samuli Laine and Tero Karras for the Hairball model, Guillermo Llaguno for the San Miguel model, the UNC for the Powerplant model, and Stanford repository for the Armadillo and Happy Buddha models.

We would also like to thank Tero Karras, Timo Aila, and Samuli Laine for releasing their GPU ray tracing framework. This research was supported by the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis) and the Grant Agency of the Czech Technical University in Prague, grant No. SGS13/214/OHK3/3T/13.

References

[1] T. Aila, S. Laine, Understanding the Efficiency of Ray Traversal on GPUs, in: Proceedings of HPG 2009, 2009, pp. 145–149.

[2] V. Havran, Heuristic Ray Shooting Algorithms, Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (November 2000).

[3] J. P. M. Masso, P. G. Lopez, Automatic Hybrid Hierarchy Creation: a Cost-model Based Approach, Computer Graphics Forum 22 (1) (2003) 5–13.

[4] S. M. Rubin, T. Whitted, A 3-Dimensional Representation for Fast Rendering of Complex Scenes, in: SIGGRAPH '80 Proceedings, Vol. 14, 1980, pp. 110–116.

[5] H. Weghorst, G. Hooper, D. P. Greenberg, Improved Computational Methods for Ray Tracing, ACM Transactions on Graphics 3 (1) (1984) 52–69.

[6] T. L. Kay, J. T. Kajiya, Ray Tracing Complex Scenes, in: D. C. Evans, R. J. Athay (Eds.), SIGGRAPH '86 Proceedings), Vol. 20, 1986, pp. 269–278.

[7] J. Goldsmith, J. Salmon, Automatic Creation of Object Hierarchies for Ray Tracing, IEEE Computer Graphics and Applications 7 (5) (1987) 14–20.

[8] S. M. Omohundro, Five Balltree Construction Algorithms, Tech. Rep. TR-89-063, International Computer Science Institute, Berkeley (Nov 1989).

[9] J. Bittner, M. Hapala, V. Havran, Fast Insertion-Based Optimization of Bounding Volume Hierarchies, Computer Graphics Forum 32 (1) (2013) 85–100.

[10] I. Wald, On fast Construction of SAH based Bounding Volume Hierarchies, in: Proceedings of the Symposium on Interactive Ray Tracing, 2007, pp. 33–40.

[11] V. Havran, R. Herzog, H.-P. Seidel, On the Fast Construction of Spatial Data Structures for Ray Tracing, in: Proceedings of IEEE Symposium on Interactive Ray Tracing 2006, 2006, pp. 71–80.

[12] T. Ize, I. Wald, S. G. Parker, Asynchronous BVH Construction for Ray Tracing Dynamic Scenes on Parallel Multi-Core Architectures, in: Proceedings of Symposium on Parallel Graphics and Visualization '07, pp. 101–108.

[13] W. Hunt, W. R. Mark, D. Fussell, Fast and Lazy Build of Acceleration Structures from Scene Hierarchies, in: Proceedings of Symposium on Interactive Ray Tracing, 2007, pp. 47–54.

[14] H. Dammertz, J. Hanika, A. Keller, Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays, Computer Graphics Forum 27 1225–1233(9).

[15] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH Construction on GPUs, Comput. Graph. Forum 28 (2) (2009) 375–384.

[16] I. Wald, Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture, IEEE Transactions on Visualization and Computer Graphics 18 (1) (2012) 47–57.

[17] J. Pantaleoni, D. Luebke, HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry, in: Proceedings of High Performance Graphics '10, 2010, pp. 87–95.

[18] K. Garanzha, J. Pantaleoni, D. McAllister, Simpler and Faster HLBVH with Work Queues, in: Proceedings of posium on High Performance Graphics, 2011, pp. 59–64.

[19] T. Karras, Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees, in: Proceedings of the EUROGRAPHICS Conference on High Performance Graphics 2012, 2012, pp. 33–37.

[20] B. Walter, K. Bala, M. Kulkarni, K. Pingali, Fast Agglomerative Clustering for Rendering, in: IEEE Symposium on Interactive Ray Tracing 2008, pp. 81–86.

[21] Y. Gu, Y. He, K. Fatahalian, G. E. Blelloch, Efficient BVH Construction via Approximate Agglomerative Clustering, in: Proceedings of High Performance Graphics, ACM, 2013, pp. 81–88.

[22] A. Kensler, Tree Rotations for Improving Bounding Volume Hierarchies, in: Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing, 2008, pp. 73–76.

[23] T. Karras, T. Aila, Fast Parallel Construction of High-Quality Bounding Volume Hierarchies, in: Proceedings of High Performance Graphics, ACM, 2013, pp. 89–100.

[24] T. Aila, T. Karras, S. Laine, On Quality Metrics of Bounding Volume Hierarchies, in: In Proceedings of High Performance Graphics, ACM, 2013, pp. 101–108.

[25] W. T. Correa, J. T. Klosowski, C. T. Silva, Visibility-Based Prefetching for Interactive Out-Of-Core Rendering, in: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG'03), 2003, pp. 1–8.

[26] C. Lauterbach, S.-E. Yoon, M. Tang, D. Manocha, ReduceM: Interactive and Memory Efficient Ray Tracing of Large Models, Comput. Graph. Forum 27 (4) (2008) 1313–1321.

[27] E. Gobbetti, D. Kasik, S. Yoon, Technical Strategies for Massive Model Visualization, in: Proc. ACM Solid and Physical Modeling Symposium, 2008, pp. 405–415.

[28] T. Karras, T. Aila, S. Laine, Understanding the Efficiency of Ray Traversal on GPUs; Google Code (2009).

Method	Build time [s]	BVH cost [-]	Stream. speed [$\frac{MTris}{s}$]	GPU primary [$\frac{MRays}{s}$]	GPU AO [$\frac{MRays}{s}$]	Build time [s]	BVH cost [-]	Stream. speed [$\frac{MTris}{s}$]	GPU primary [$\frac{MRays}{s}$]	GPU AO [$\frac{MRays}{s}$]	Build time [s]	BVH cost [-]	Stream. speed [$\frac{MTris}{s}$]	GPU primary [$\frac{MRays}{s}$]	GPU AO [$\frac{MRays}{s}$]
	Sibenik, 80k triangles					Conference, 283k triangles					Armadillo, 307k triangles				
SAH	0.44	82.3	n/a	137	191	1.93	130	n/a	124	198	1.98	86.3	n/a	159	86.5
Median	0.06	391	n/a	18.7	27.3	0.20	842	n/a	14.4	30.8	0.24	144	n/a	93.8	61.1
as is															
Incr	0.11	80.2	0.68	105	140	0.68	119	0.41	113	192	0.77	101	0.39	125	74.4
IncrU	0.12	**73.8**	0.66	**127**	**176**	0.68	**109**	0.41	**123**	215	0.87	**97.3**	0.35	130	**75.5**
IncrUP	0.10	82.0	0.78	99.8	144	0.49	121	0.56	97.6	178	0.70	98.3	0.43	**132**	75.5
IncrUPB	0.04	83.7	**1.96**	93.2	136	0.13	133	**2.11**	116	**216**	0.58	111	**0.52**	111	68.3
view direction															
Incr	0.24	85.8	0.33	74.6	114	1.00	132	0.28	96.1	179	1.62	273	0.18	56.8	38.8
IncrU	0.22	**73.9**	0.35	**125**	**156**	0.92	109	0.30	103	203	1.17	134	0.26	91.1	61.2
IncrUP	0.18	74.8	0.42	101	142	0.72	**108**	0.39	**112**	**216**	0.88	133	0.34	90.0	61.9
IncrUPB	0.04	96.0	**1.67**	58.3	116	0.19	128	**1.42**	72.4	143	0.22	**126**	**1.38**	**102**	**64.6**
random															
Incr	0.24	79.4	0.33	116	148	1.30	116	0.21	125	221	0.96	94.9	0.31	133	77.9
IncrU	0.29	71.7	0.27	**136**	**181**	1.45	**105**	0.19	**132**	237	1.12	**94.1**	0.27	**138**	**78.2**
IncrUP	0.23	**71.4**	**0.33**	125	179	1.13	105	**0.25**	121	**238**	0.93	94.3	**0.32**	136	77.9
IncrUPB	0.25	91.6	0.32	100	133	1.37	139	0.20	68.0	117	1.08	105	0.28	114	73.1
	HappyBuddha, 1,087k triangles					SodaHall, 2,169k triangles					Hairball, 2,880k triangles				
SAH	8.91	165	n/a	355	82.6	22.5	217	n/a	113	156	24.1	1415	n/a	13.6	36.9
Median	0.82	276	n/a	203	44.9	1.88	1396	n/a	8.11	8.96	2.42	2447	n/a	8.18	21.2
as is															
Incr	2.63	346	0.41	162	42.5	3.84	204	0.56	84.2	116	6.69	1517	0.43	9.23	25.6
IncrU	2.35	**230**	0.46	**227**	**56.2**	3.55	**183**	0.61	75.1	**157**	6.19	**1460**	0.46	**11.2**	**29.7**
IncrUP	1.76	242	0.61	210	52.2	2.90	224	0.74	67.2	95.9	5.18	1908	0.55	7.60	22.8
IncrUPB	1.56	271	**0.69**	170	49.9	0.76	229	**2.85**	**86.2**	113	1.08	2115	**2.65**	7.03	16.1
view direction															
Incr	6.13	457	0.17	120	36.0	8.52	220	0.25	102	134	18.8	1772	0.15	8.68	22.7
IncrU	4.49	243	0.24	**233**	**55.0**	8.16	**188**	0.26	**121**	155	18.0	1571	0.15	10.4	27.1
IncrUP	3.39	**240**	0.32	226	55.0	6.54	189	0.33	81.8	**158**	14.1	**1569**	0.20	**10.7**	**27.6**
IncrUPB	1.39	289	**0.77**	148	49.6	2.05	238	**1.05**	38.0	87.8	8.08	2601	**0.35**	4.43	18.8
random															
Incr	4.53	184	0.24	**298**	72.1	12.5	198	0.17	112	135	17.7	1431	0.16	11.8	31.5
IncrU	5.49	**181**	0.19	294	**73.1**	14.5	**183**	0.14	**115**	**175**	20.3	**1424**	0.14	**12.0**	**31.7**
IncrUP	4.21	183	**0.25**	291	72.4	11.4	185	**0.19**	107	157	15.8	1424	**0.18**	11.7	31.2
IncrUPB	4.85	194	0.22	266	67.8	13.1	229	0.16	62.0	101	21.8	1853	0.13	9.71	26.4
	Pompeii, 5,646k triangles					SanMiguel, 7,881k triangles					PowerPlant, 12,749k triangles				
SAH	46.7	253	n/a	24.7	36.4	107	181	n/a	44.0	95.5	209	116	n/a	141	75.1
Median	4.27	767	n/a	8.59	12.9	7.96	1278	n/a	4.32	8.62	14.6	661	n/a	8.82	9.44
as is															
Incr	11.4	266	0.49	20.8	36.0	20.3	177	0.38	40.3	80.6	34.7	120	0.36	35.4	74.3
IncrU	10.6	**231**	0.53	**24.5**	**42.4**	17.9	**158**	0.44	**48.6**	**99.3**	27.5	**104**	0.46	**139**	**82.0**
IncrUP	7.97	258	0.70	20.8	34.3	13.3	172	0.59	34.4	84.2	20.3	118	0.62	101	61.4
IncrUPB	2.13	272	**2.64**	20.2	35.3	4.92	192	**1.59**	34.0	69.3	4.63	117	**2.75**	87.6	64.6
view direction															
Incr	27.7	274	0.20	19.7	34.2	49.7	212	0.15	25.4	58.9	121	132	0.10	96.9	59.5
IncrU	25.8	**240**	0.21	**23.0**	**38.4**	46.7	**165**	0.16	38.9	84.1	114	**107**	0.11	**126**	**78.3**
IncrUP	19.3	240	0.29	22.5	36.4	36.3	166	0.21	**41.7**	**87.2**	93.3	108	0.13	118	77.2
IncrUPB	4.53	348	**1.24**	15.7	27.4	16.3	205	**0.48**	26.5	58.4	44.8	149	**0.28**	60.5	40.7
random															
Incr	41.1	241	0.13	23.5	38.7	58.8	169	0.13	33.5	86.9	115	107	0.11	131	76.6
IncrU	48.8	234	0.11	24.1	37.6	69.2	154	0.11	43.5	101	136	**102**	0.09	**149**	85.8
IncrUP	35.7	**233**	**0.15**	**24.9**	**40.2**	52.3	**153**	**0.15**	**45.6**	**102**	93.5	103	**0.13**	140	**86.5**
IncrUPB	46.6	313	0.12	17.4	32.2	64.6	178	0.12	35.1	78.7	128	130	0.09	60.0	56.0

Table 1: Results of the incremental BVH build. The lowest BVH costs and the highest streaming and rendering speeds for the given scene and the stream order are highlighted.