STBVH: A Spatial-Temporal BVH for Efficient Multi-Segment Motion Blur

Sven Woop
Intel Corporation

Attila T. Áfra
Intel Corporation

Carsten Benthin
Intel Corporation

Llama *Barbershop* *Train*

Figure 1: Scenes with multi-segment motion blur from Blender Institute Open Movies rendered using our proposed STBVH acceleration structure. Compared to building separate BVHs for each global time segment, the STBVH reduces memory consumption for motion blurred geometry by 1.2–2.6×, while the rendering performance is only up to 4% lower for these scenes.

ABSTRACT

We present the *STBVH*, a new approach for rendering multi-segment motion blur using a bounding volume hierarchy (BVH) that stores both spatial linearly interpolated bounds and temporal bounds. The approach is designed for different number of time steps per mesh or object. While separating the individual meshes using standard partitioning techniques, it performs temporal splits for locations with large or curved motion inside the meshes. Our approach uses a modified motion blur surface area heuristic (MBSAH) that calculates probabilities in the presence of spatial-temporal bounds and works on linear motion segments of primitives rather than on full motion curves. We show that our approach is able to handle challenging scenes with varying degrees of motion blur per mesh, using significantly less memory and having competitive rendering performance compared to building separate linear motion blur BVHs per global time segment.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**; *Visibility*;

KEYWORDS

ray tracing, multi-segment motion blur, BVH

ACM Reference format:
Sven Woop, Attila T. Áfra, and Carsten Benthin. 2017. STBVH: A Spatial-Temporal BVH for Efficient Multi-Segment Motion Blur. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 8 pages.
https://doi.org/10.1145/3105762.3105779

HPG '17, July 28-30, 2017, Los Angeles, CA, USA

1 INTRODUCTION

The preferred method for rendering movies today is Monte Carlo path tracing [Christensen and Jarosz 2016], which relies on ray tracing to directly sample the rendering equation to compute photo-realistic images. Tracing rays is typically the most time consuming part of this process. For ray tracing animated content, the rendering of motion blur effects is essential for achieving high visual quality, but it is also very challenging in both rendering time and memory consumption. Monte Carlo path tracing handles motion blur by stochastically integrating over the camera's shutter time; i.e., any ray being traced has a random "time" associated with it. This in turn complicates the underlying ray tracing kernel as it has to find the proper ray-primitive intersection for a given time stamp. For objects that are moving quickly the respective primitives' locations can vary significantly over the time the shutter is open, making it harder to find the correct ray-primitive intersection for a specific point in time.

Supporting only linear motion blur, where each primitive is specified through a start- and end-position and the exact position for a specific time is obtained through linear interpolation, simplifies the problem. In fact, many systems already have support for linear motion blur, however, for very quickly moving objects that do not move in a straight trajectory (anything that is rotating quickly, such as a rotor, waving arms, a carousel, etc.), simple linear interpolation does not provide an adequate approximation (see Figure 2). These scenarios require multi-segment motion blur, where each primitive (or vertex) follows along a path of N linear line segments, or along a spline-based curve defined by multiple control points.

In this paper we present the *Spatial-Temporal Bounding Volume Hierarchy (STBVH)*, which efficiently supports multi-segment motion blur, while minimizing memory usage and offering similar rendering performance to simple linear motion blur BVHs.

(a) Linear motion blur (b) Multi-segment motion blur

Figure 2: A closeup of the *Train* scene showing quickly rotating wheels rendered with (a) linear motion blur with 2 time steps and (b) multi-segment motion blur with 17 time steps. Linear motion blur is not capable of capturing the complex non-linear motion of the wheels and thus causes incorrect blurring.

2 PREVIOUS WORK

In the past, researchers investigated different approaches for handling motion blur in ray tracing-based rendering. Linear motion blur simplifies the problem to just two time steps with linear interpolation of vertices inbetween. The naive approach of just using a standard BVH build over the moving primitives suffers from performance issues due to overlapping bounds in particular if the motion gets larger than the primitive size. This issue can be solved by using linearly interpolated bounds [Christensen et al. 2006; Hanika et al. 2010; Hou et al. 2010] which can tighly bound linear motion, however, these linear bounds cannot handle fast curved motion efficiently as described above.

In the case of multi-segment piecewise linear motion, a straightforward extension of the linearly interpolated approach is to use multiple linear motion BVHs for a sufficiently large *global* number of time segments, which together cover all motion in the scene. Ideally, the global number of time segments should be equal to the least common multiple of the numbers of time segments per object. However, if the time segments are not restricted to powers of two (which is common), this would be impractical, so using the maximal number of time segments (or even less) is a more reasonable choice, at the cost of lower tree quality. Unfortunately, this approach has two major issues. First, it requires N independent linear motion blur BVHs, which is quite expensive in terms of memory consumption. This is in particular problematic if different objects have different numbers of time steps, as one would have to build potentially as many BVHs as the maximal number of time segments. Second, since each time segment essentially has its own BVH, ray packet approaches, which trace small packets with multiple rays simultaneously, are problematic as different rays might have different time values assigned. In this case, the rays will quickly diverge into different BVHs, lowering the efficiency of these packet techniques.

Instead of building separate BVHs over sufficiently many time segments, an alternative would be to use a single, shared BVH topology for all time segments, storing multiple bounding volumes per node [Grünschloß et al. 2011]. In this approach each node stores the maximal number of bounding volumes with respect to its children (even if the amount of motion is low), which again results in increased memory consumption. Also, this approach is typically limited to power-of-two time segments, as this simplifies the implementation. Furthermore, a single topology cannot be optimal for each time step. For example, if two primitives are close together in one time step, putting them into a leaf node seems optimal, while the same primitives may be far apart in a second time step where this leaf would have a large surface area and cause low rendering performance.

A different approach relies on extending spatial data structures to a 4D structure, with time being the 4^{th} dimension. An extension of k-d trees to 4D was introduced by Olsson [Olsson 2007] by adding temporal splits. The advantage of this approach is that it can use temporal splits only where necessary. However, k-d trees cannot efficiently handle linear motion in the first place, therefore requiring many temporal splits and thus reducing the efficiency of the data structure. Similarly, BVHs have been extended to a 4D data structure [Glassner 1988] storing bounds using slabs in 12 space-time directions. In contrast, our approach supports tighter 4D bounding by using linearly interpolated bounds, which are oriented along the direction of motion, and not along fixed directions. Ray classification has also been extended to 4D, but its memory requirements are too large for practical purposes [Quail 1996].

Rather than rendering one frame over the entire shutter time, the renderer could render separate frames for sufficiently many time segments, and average/blend those partial frames, while within a single frame it could rely on linear motion blur only. However, this approach introduces lots of complexity into the renderer itself, and further limits additional techniques, such as post-frame filtering, interactive editing, and interactive previews. It also requires the renderer to re-build a different BVH for each of the N frames, and to re-compute all other per-frame data for each such time region (which is often unnecessary).

Re-building the BVH from scratch for each frame can be avoided by building only a single BVH (e.g. for the first frame) and then refitting it for each frame [Wald et al. 2007]. However, this often breaks down for complex deformations and topology changes. The quality of the per-frame BVHs can be improved by using the T-SAH [Bittner and Meister 2015], which is an extension of the traditional SAH to sequences of animated frames. First, a BVH is built for one frame, and then it is iteratively optimized for the entire sequence using the T-SAH, which is a weighted average of the SAH costs of the individual frames. However, refitting the BVH for each frame is still necessary. Further, if one object of the scene requires many time steps, the entire scene needs to be processed for a large number of time segments multiple times.

3 THE SPATIAL-TEMPORAL BVH (STBVH)

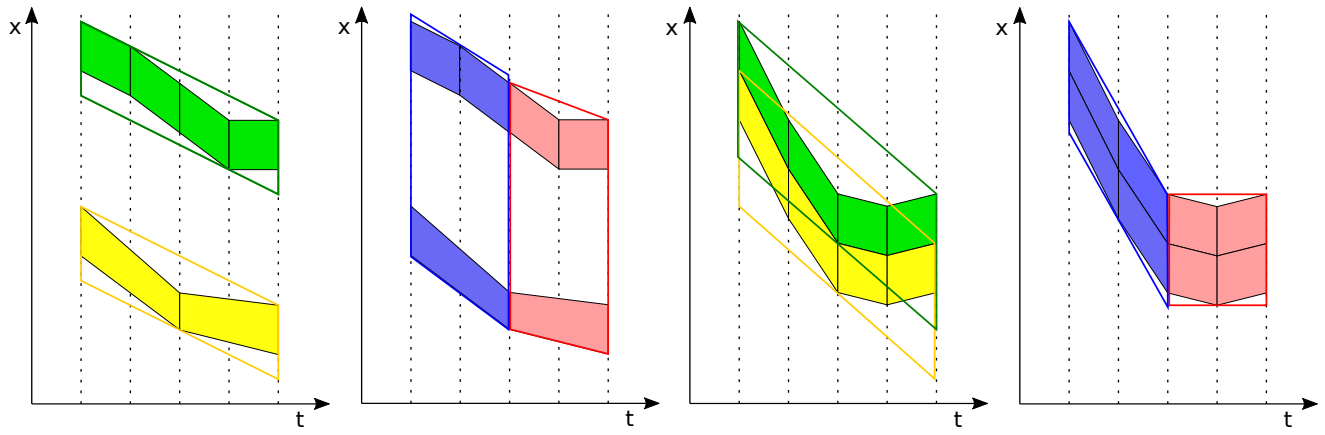
Our approach is based on a BVH that introduces temporal bounds and a build algorithm that can perform temporal splits to simplify the motion of a subtree.

The data structure encodes the motion for the normalized shutter time $[0, 1]$ and we only accept ray timestamps in this normalized range.

3.1 Data Structure Layout

The fundamental data structure to our approach is an N-ary bounding volume hierarchy (BVH) where each node stores bounds and pointers for up to N children. Our BVH has three node types:

(1) **Spatial Nodes** store spatial *linear bounds* for each child.



(a) Example 1: Object partitioning (b) Example 1: Temporal partitioning (c) Example 2: Object partitioning (d) Example 2: Temporal partitioning

Figure 3: Object partitioning and temporal partitioning for some example scenes of moving 1D lines. The diagrams show the time axis to the right and spatial axis upwards. The trapezoid shapes correspond to the linear motion of the 1D lines moving from one time step to the next and are counted by the MBSAH heuristic through the $|X|_s$ factor. The area of the drawn linear bounds correspond to the $SA'(X) \cdot T(X)$ factor of the heuristic in the 1D case. The first example shows two separated objects with different number of time steps that move downwards. The best object split (a) produces tight linear bounds while the best temporal split (b) produces loose linear bounds. In this example, our heuristic would choose the object split. The second example shows some curved motion of an object. The best object split (c) produces large linear bounds with overlaps, while the best temporal split (d) produces tight linear bounds and thus would get chosen by our heuristic.

(2) **Spatial-Temporal Nodes** store spatial *linear bounds* and temporal bounds for each child.
(3) **Leaf Nodes** store a list of primitives.

Linear bounds refer to a pair of axis aligned bounding boxes (AABBs) whose linear interpolation to time t bound the geometry at that time. A spatial-temporal node stores temporal bounds $T \subset [0,1]$ and spatial bounds (B_0, B_1), while $B_t = (1-t) \cdot B_0 + t \cdot B_1$ bounds the geometry represented by the child for each time $t \in T$. Note that we directly use the global time t to interpolate the bounds and do not normalize the time to the time range T. This way we can always directly use the time stored inside the ray to interpolate linear bounds stored in the BVH, no matter which time range is active in the current subtree.

We choose to use linear bounds in our data structure as they have already proven to handle linear motion blur well [Hou et al. 2010]. They in particular allow us to bound the typical case of linear motion very tightly, but can also be used to bound more general motion.

We added spatial-temporal nodes to the data structure (in contrast to temporal-only nodes), as this allows us to perform object and temporal partitioning mixed arbitrarily during the BVH build process without loosing culling efficiency. A subtree of the BVH represents only the geometry for the time range specified inside the spatial-temporal node. This allows the data structure to shrink the time range in areas where it is difficult to bound motion, e.g. in case of curved motion. Temporal bounds of neighboring subtrees may overlap in our data structure, however, our BVH build algorithm currently does not produce temporal overlaps. When going down the BVH the temporal bounds of a lower level are always a subset of the temporal bounds on a higher level, thus temporal bounds always shrink.

As an optimization we kept spatial-only nodes in our data structure, because for large parts of the BVH no temporal partitioning is performed and temporal bounds are not required (e.g., if motion in the current time range is small or very linear).

Leaf nodes store primitives that can be intersected for any time in the current time range. These primitives are typically stored with respect to piecewise linear motion. While the data structure can also handle non-equidistant piecewise linear motion or even higher order motion, we will assume equidistant piecewise motion from now on. Thus while each primitive can have a different number of time segments, we assume that the time segments of a primitive are of equal length.

We use an N-ary BVH because this type of data structure allows for efficiently exploiting SIMD instructions of modern CPUs to achieve high performance [Wald et al. 2014]. As branching factor N we either use 4 or 8 depending on the available SIMD width of the underlying architecture.

3.2 Motion Blur SAH (MBSAH)

Standard top-down BVH construction techniques use the local greedy surface area heuristic (SAH) [Goldsmith and Salmon 1987; Wald and Havran 2006] to decide whether and how to partition a set of primitives. This heuristic uses probabilities of hitting sets of primitives to estimate the cost of leaf creation or performing a split the following way:

$$C_{leaf}(X) = |X| \cdot C_I$$
$$C_{split}(X, X_0, X_1) = C_T + P(X_0|X) \cdot C_{leaf}(X_0) + P(X_1|X) \cdot C_{leaf}(X_1)$$

Here X denotes a set of primitives and the leaf creation cost function C_{leaf} estimates the cost of sequentially intersecting all primitives, where $|X|$ is the number of primitives in X, and C_I is a constant estimating the intersection cost of one primitive. In the split cost function C_{split}, the term $P(Y|X)$ denotes the conditional probability of a ray that hits the bounding box of X also hits the bounding box of a subset Y of the primitives, and C_T estimates the cost of one traversal step.

The probability $P(Y|X)$ is calculated using the ratio of the surface area of the axis aligned bounds of Y and the surface area of the enclosing bounding box X:

$$P(Y|X) = \frac{SA(Y)}{SA(X)}$$

where $SA(X)$ is the surface area of the bounding box of X.

For our *motion blur surface area heuristic (MBSAH)* we attach a time range to each primitive and count the total number $|X|_s$ of active *primitive segments* of the primitives. A primitive segment represents the shape of the linear motion of a primitive from one time step to the next, and it is considered active in a time range if that range overlaps its own time range.

As we use spatial-temporal bounds we have to adjust the probability of hitting these bounds. First a ray may also miss spatial-temporal bounds if its time does not fall into the time range of these bounds. Second, as we use linear bounds during construction and in our data structure, we need to approximate the time-averaged surface area of these linear bounds. Thus as probability the MBSAH uses:

$$P(Y|X) = \frac{SA'(Y)}{SA'(X)} \cdot \frac{T(Y)}{T(X)}$$

Here $SA'(X)$ is the surface area of the center-time bounds of the linear bounds of X, the merged time range of X is used to calculate these center-time bounds, and $T(X)$ is the size of this merged time range. Note that using the average surface area observed by a random ray would be more accurate, but this surface area approximation showed no practical decrease in BVH quality.

After multiplying the cost functions by $SA'(X) \cdot T(X)$ to avoid expensive divisions (we can do this because we do not need the absolute cost values), our final MBSAH looks like the following:

$$\hat{C}_{leaf}(X) = SA'(X) \cdot T(X) \cdot |X|_s \cdot C_I$$

$$\hat{C}_{split}(X, X_0, X_1) = SA'(X) \cdot T(X) \cdot C_T + \hat{C}_{leaf}(X_0) + \hat{C}_{leaf}(X_1)$$

As we calculate the surface area of the linear bounds, this heuristic is a better estimate of the traversal cost, compared to previous approaches that calculate SAH costs based on axis aligned bounds to build linear motion BVHs.

Also, as we count active primitive segments, this SAH calculation makes meshes with a large number of active time segments more expensive than a mesh with a smaller number of active time segments. It further makes splitting at times that fall inside a time segment more expensive, as this primitive segment now counts on both sides of the split. For this reason, local minima for temporal splits are typically found at discrete locations between time segments.

The heuristic can be used to evaluate *temporal splits* that split the merged time ranges of the primitives into two halves (and partitioning primitives while properly adjusting their time range), or to *object splits* that partition the primitives using some spatial criteria but keep the time ranges unchanged. Some examples of the heuristic applied to object and temporal splits are illustrated in Figure 3.

A challenging case for a motion blur SAH is a geometry where neither object nor temporal splits can reduce the spatial bounding box (e.g., small triangles uniformly distributed along a circle and rotating around the circle twice). In such a scenario temporal splits have to get chosen to shrink the time range until the primitive motion is sufficiently small to perform object splits. Our heuristic robustly chooses temporal splits in this scenario. The motion blur SAH uses the probability of hitting the spatial-temporal bounds, and these spatial-temporal bounds *always* shrink when performing a temporal split (as the time range gets split). Thus temporal splits are always *good* in shrinking the spatial-temporal bounds, and will always get selected when an object split produces *bad* SAH cost. Note that our decision to count segments rather than primitives plays an important role here, because the set of active primitives is typically not partitioned well using a temporal split. The reason for this is that primitives are most of the time active over the entire shutter time.

A second case to consider is of spatially separated motion blurred objects (see Figure 3) . For such a case we intend to first spatially separate the objects using object splits, and then use temporal splits inside the objects to simplify the motion. The object splits typically reduce the surface area sufficiently (e.g., cut a cubic bounding of area 1 into two bounds with area $\frac{2}{3}$), while temporal splits may partition time segments well and cut the temporal bounds into two halves (e.g., cut unit temporal bounds of size 1 into two temporal bounds of size $\frac{1}{2}$). This calculation shows that temporal splits are favored in this situation (essentially due to the area of the splitting plane). To counteract this effect we penalize temporal splits slightly by a factor $\lambda = 1.25$ when selecting between object and temporal splits. This makes both splits about equally expensive in this situation. Using this factor does not change rendering performance much, but helps reducing BVH size, as the temporal splits tend to replicate primitive segments (that fall onto the splitting time) on both sides of the split. Note that the temporal split will also get penalized if it cannot partition the time segments of the primitives well (e.g., if there are many objects with only a single time segment), which further increases the likelyhood that object splits are performed in this situation.

3.3 Construction

We perform a top-down construction of our data structure using the modified MBSAH. Algorithm 1 shows the pseudocode of the recursive construction algorithm which builds the subtree for a set of *build primitives* X for the time range T, and returns the constructed node N and linear bounds B. During construction we evaluate the SAH heuristic to either partition the set of *build primitives* X or split the current *time range* T.

A *build primitive* is a data structure that represents the entire motion path of a single primitive. It references multiple primitive segments for one primitive, which together define the entire animation during the shutter time. A build primitive is active for a time

Algorithm 1 STBVH construction. The BuildNode function recursively builds the subtree for a set of build primitives X for time range T, and returns the constructed node N and linear bounds B.

```
1:  function BuildNode(X, T)
2:      c_leaf ← CalculateLeafCost(X, T)
3:      c_objectSplit ← FindObjectSplit(X, T)
4:      c_temporalSplit ← ∞
5:      if c_objectSplit ≥ θ · c_leaf then
6:          if |T| ≥ MinTimeSegmentSize(X) then
7:              c_temporalSplit ← FindTemporalSplit(X, T)
8:          end if
9:      end if
10:     c_best ← min(c_leaf, c_objectSplit, λ · c_temporalSplit)
11:     if c_best = c_leaf then
12:         N ← LeafNode(X)
13:         return (N, CalculateLinearBounds(X, T))
14:     else if c_best = c_objectSplit then
15:         (X_0, X_1) ← PerformObjectSplit(X, T)
16:         (N_0, B_0) ← BuildNode(X_0, T)
17:         (N_1, B_1) ← BuildNode(X_1, T)
18:         N ← SpatialNode(N_0, B_0, N_1, B_1, T)
19:         return (N, MergeLinearBounds(B_0, B_1))
20:     else
21:         (X_0, T_0, X_1, T_1) ← PerformTemporalSplit(X, T)
22:         (N_0, B_0) ← BuildNode(X_0, T_0)
23:         (N_1, B_1) ← BuildNode(X_1, T_1)
24:         N ← SpatialTemporalNode(N_0, B_0, T_0, N_1, B_1, T_1)
25:         return (N, CalculateLinearBounds(X, T))
26:     end if
27: end function
```

range if any of its primitive segments is active for that time range. The resolution of the motion may be different for each primitive, thus each build primitive may refer to a different number of primitive segments. Primitives may also start and end to *live* during the shutter time, which just shrinks the time range the corresponding build primitives are considered to be active in.

Each build primitive stores a primitive ID, local linear bounds for the current time range, the number of primitive segments active during that time range, and the total number of primitive segments of that primitive. Note that we do not have to store the current time range as this is known implicitly during construction and that the linear bounds stored are local, thus they store the bounds at the beginning and the end of the time range. These local linear bounds can be merged tightly, e.g. for the SAH binning phase, by just merging the bounds for the begin and end times.

The builder first tests whether a standard *object split* gives low SAH cost (lines 2–5). Therefore we bin the build primitives using the center of their linear bounds interpolated to the current center time, and evaluate the SAH for all three spatial dimensions. This can be done efficiently as the linear bounds and the number of active primitive segments are directly stored inside the build primitives X. We consider the object split successful if $c_{objectSplit} < \theta \cdot c_{leaf}$, where $c_{objectSplit}$ and c_{leaf} are the object split and leaf costs, respectively. We use $\theta = 0.5$, which essentially means that the object split should at least cut surface areas in half.

Otherwise, we test a *temporal split* of the current time range T into T_0 and T_1, but only if T is not already smaller than the minimal time segment size over all current build primitives (lines 6–8). For splitting time we essentially choose the center time, and align it to a discrete time segment boundary of the build primitive that has the most time segments. After this time has been calculated, we iterate over all current build primitives and calculate the SAH for the selected split. This task is quite expensive, as we need to recalculate linear bounds and number of overlapping time segments for the time ranges T_0 and T_1 for each build primitive.

Next, we select the best of c_{leaf}, $c_{objectSplit}$, and $\lambda \cdot c_{temporalSplit}$ and perform the corresponding split or create a leaf (lines 10–26). λ essentially makes temporal splits slightly more expensive and we choose $\lambda = 1.25$ in our implementation as described above. Object splits can partition the build primitive array in-place, while temporal splits typically output two build primitive arrays of the same size (the size can also shrink as primitives may be inactive for some time range). We re-use previously allocated memory to store the first array, and allocate new memory for the second array.

The just described algorithm builds a binary BVH. To fill N-wide nodes of an N-ary BVH, we iteratively split the child having the largest estimated surface area until the N-wide node becomes full.

The builder returns the local linear bounds of the generated node from the recursion. This way, spatial nodes can just use these bounds to create the final node, and return the merged bounds (lines 18–19). When creating the node, the local linear bounds have to be converted to global linear bounds to be stored inside the BVH nodes. This can easily be achieved by interpolating the local linear bounds to the global start time 0 and end time 1.

Spatial-temporal nodes also use the bounds returned by the recursive construction to create the node (line 24). Merging these spatial-temporal bounds of different time ranges is possible, but

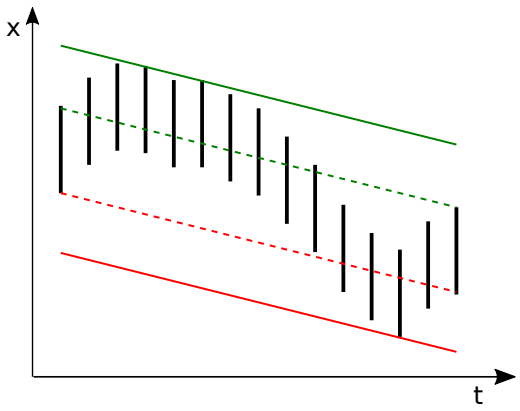


Figure 4: This figure illustrates how we calculate linear bounds for the motion of a primitive. In this example, a 1D line segment is moving in time. We calculate initial upper bounds by using the upper bounds at the start and end times as initial upper bounds (dashed green line) and move these bound upwards until they bound the primitive at all time steps (green line). Similarly, we adjust initial lower bounds (dashed red line) downwards to calculate the lower bounds (red line).

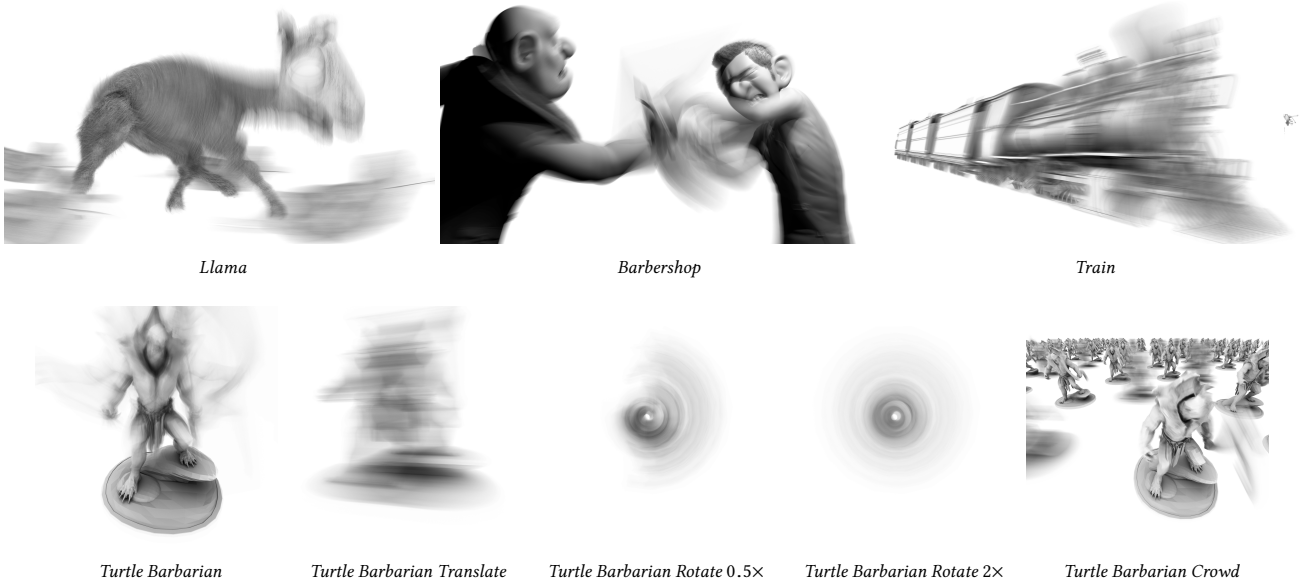


Figure 5: Test scenes with varying degrees of motion blur used for the performance measurements. See Table 1 for primitive and time step counts.

produces quite coarse bounds. To obtain tight linear bounds for the spatial-temporal nodes, we therefore iterate over all current primitives, recalculate their linear bounds for the node's time range, and merge these linear bounds (line 25).

We calculate linear bounds for a set of primitives as illustrated in Figure 4. To calculate linear upper bounds for the x dimension, we choose the upper bounds of x for the first and last time step as initial linear bounds. Interpolating these two values to intermediate times yields a line that typically is no upper bound for each time step yet. To fix this we just add some constant (the maximal error) to the linear bounds which moves the line upwards such that it is an upper bound for all time steps. Other dimensions and lower bounds are calculated similarly.

3.4 Ray Traversal

The ray traversal through the STBVH works similar to a standard BVH traversal. Ray-bounds intersection tests are performed recursively to determine which children are intersected, and these children are processed in an intersection-distance-based order. Once a leaf is reached, the geometry contained in the leaf is intersected.

Intersecting the linear bounding boxes stored in the nodes works by linearly interpolating them to the time t stored in the ray. If the nodes also store time intervals, we additionally check if the ray falls into the time interval, to decide whether we need to traverse the subtree at all.

Some care has to be taken if the time of a ray falls exactly onto a boundary of a time range, to avoid entering two time ranges. To solve this issue, time ranges $[t0, t1[$ are always treated to be open to the right and we increase the end time 1 by an epsilon when stored inside the BVH to properly handle a ray with a time of 1.

Scene	Primitive Groups	
	Time Steps	Primitives
Llama	3	7.0M
	9	1.7M
Barbershop	3	1.4M
	5	2.8M
	9	3.9M
Train	3	0.3M
	17	2.0M
Turtle Barbarian	15	0.1M
Turtle Barbarian Translate	6	0.1M
Turtle Barbarian Rotate 0.5×	9	0.1M
Turtle Barbarian Rotate 2×	33	0.1M
Turtle Barbarian Crowd	2	7.5M
	6	2.8M
	15	0.1M

Table 1: The test scenes consist of primitives (triangles and line segments) that may have different number of time steps. This table groups the primitives by the numbers of time steps they have, and lists for each group the number of primitives that are stored at the respective time resolution.

4 RESULTS

We implemented our approach in the Embree Ray Tracing Kernels framework [Wald et al. 2014], and compared it against the existing multi-segment motion blur implementation in Embree, which builds independent linear motion blur BVHs for a global number of time segments. The number of BVHs is equal to the maximal number of time segments of the objects in the scene. This enables very high ray traversal performance, but can result in excessive memory

Scene	Size (MB)			SAH Cost			Build Speed (Mprim/s)			Render Speed (Mray/s)		
	BVH	STBVH	Ratio	BVH	STBVH	Ratio	BVH	STBVH	Ratio	BVH	STBVH	Ratio
Llama	2631.5	1019.5	0.39×	20.2	21.3	1.05×	5.3	7.41	1.40×	17.0	16.6	0.98×
Barbershop	2775.4	1873.2	0.67×	46.8	48.3	1.03×	5.47	3.74	0.68×	25.3	25.6	1.01×
Train	1788.4	1529.7	0.86×	2	2.1	1.05×	2.83	1.73	0.61×	43.3	41.4	0.96×
Turtle Barbarian	68.7	68	0.99×	8.5	9.1	1.07×	2.17	1.45	0.67×	62.0	57.2	0.92×
Turtle Barbarian Translate	24.8	13.3	0.54×	11	11.7	1.06×	6.13	5.32	0.87×	79.0	79.6	1.01×
Turtle Barbarian Rotate 0.5×	39.1	39.6	1.01×	8.6	6.9	0.80×	4.05	2.6	0.64×	121.6	115.2	0.95×
Turtle Barbarian Rotate 2×	156.4	158.4	1.01×	5.9	7	1.19×	0.99	0.73	0.74×	117.4	106.4	0.91×
Turtle Barbarian Crowd	6898.9	786.1	0.11×	11	11.1	1.01×	2.75	11.45	4.16×	40.1	49.5	1.24×

Table 2: Total BVH size (including BVHs for triangles and line segments) in MB, total SAH cost, build performance in million primitives per second (Mprim/s), and rendering performance using diffuse path tracing in million rays per second (Mray/s) for separate BVHs for the maximal number of time segments (BVH) and our STBVH. Only primitives with motion blur were included in the scenes. The rendering resolutions were 1920×1080 and 1440×1440 pixels.

usage if there is high variation in the number of time steps per object. Both approaches use indexed primitives, so the primitive vertices themselves are stored only once and are shared across the per-segment BVHs.

Embree currently builds separate BVHs for motion blurred and static (non-motion blurred) geometry, and for each primitive type. We do the same for the STBVH, however, this is not a limitation of our approach, but of the Embree framework itself. To isolate the performance of motion blur, we disabled all non-motion blurred geometry for our measurements.

Our tests scenes include both real-world and synthetic scenes (see Figure 5). *Llama* and *Train* are scenes from the *Caminandes 3* animated movie, and *Barbershop* is a scene from the *Agent 327* animated movie. The *Llama* and *Barbershop* scenes consist of triangles and line segments (for hair), and the *Train* scene has only triangles. The objects in all these scenes have different number of time steps and all static geometry has been removed. *Turtle Barbarian* is a triangle mesh with a complex, non-linear animation, *Turtle Barbarian Translate* is the same model but with simple linear translation motion and no deformation, *Turtle Barbarian Rotate* 0.5× and 2× are versions with half- and double rotation of the model, and *Turtle Barbarian Crowd* contains many animated, non-instanced copies of the model with varying number of time steps.

The measurements were performed on an Intel® Xeon® E5-2699 v4 workstation (*Broadwell* microarchitecture, 22 cores, 2.2 GHz) with 32 GB RAM. The code was compiled with Intel® C++ Compiler 17.0.2, and the benchmark was run under Linux.

Table 2 shows that for many scenes the STBVH significantly reduces memory usage compared to simple per-segment BVHs. The rendering performance is typically roughly the same or only slightly lower, but in certain extreme cases (like *Turtle Barbarian Crowd*) it can be even higher. The amount of possible memory reduction depends on whether the objects have different number of time steps and on the complexity of the motion as well.

For the three movie scenes, *Llama*, *Barbershop*, and *Train*, which have mixed number of time steps, the memory usage is reduced by 1.17–2.58×. For the *Turtle Barbarian* scene and its two rotated versions, where all primitives have the same number of time steps and the motion is non-linear, the memory usage is about the same. Thus, in such cases the MBSAH has no benefits compared to per-segment BVHs, but importantly, it does not produce bigger trees either. *Turtle Barbarian Translate* also has uniform time resolution but the motion is linear, which is detected by the MBSAH, reducing the memory usage by almost 2×. The biggest improvements were measured for the *Turtle Barbarian Crowd* scene, where most objects have few time steps but one object (the main character) has much more time steps than the others. This object significantly increases the number of per-segment BVHs, which makes that approach very inefficient. The STBVH handles this case well, requiring 9× less memory.

Despite the significant BVH size reductions, the total SAH cost of the STBVH is worse by only a few percent for most scenes. Using a simple single-ray diffuse path tracer with minimal shading, the rendering performance of the STBVH vs. per-segment BVHs ranges between 0.91–1.24×. Therefore, our approach achieves, on average, roughly the same or only slightly lower ray traversal performance, while significantly reducing memory usage.

The main drawback of our method is that temporal partitioning has a significant computational and memory overhead, which also makes object partitioning more expensive because of the additional bookkeeping throughout the entire build process. This overhead causes a decrease of build performance by up to 1.64×. However, if the MBSAH reduces the number of nodes and thus partitioning

Scene	Peak Build Memory (MB)		
	BVH	STBVH	Ratio
Llama	2974.6	1847.0	0.62×
Barbershop	3106.7	3614.3	1.16×
Train	1953.4	2358.5	1.21×
Turtle Barbarian	86.9	143.6	1.65×
Turtle Barbarian Translate	32.7	24.5	0.75×
Turtle Barbarian Rotate 0.5×	46.3	91.2	1.97×
Turtle Barbarian Rotate 2×	187.5	321.3	1.71×
Turtle Barbarian Crowd	7525.9	1559.2	0.21×

Table 3: The peak memory consumption during BVH build for separate BVHs for the maximal number of time segments (BVH) and our STBVH.

steps by a large enough amount, building the STBVH can be actually *faster*. This can be observed for the *Llama* and *Turtle Barbarian Crowd* scenes, where our building algorithm is faster by 1.4–4.16×.

Table 3 shows the peak memory consumption during BVH build. As temporal splits double the build primitive array size, the peak memory consumption of the STBVH build can be higher than building multiple BVHs per time segment sequentially. For example, the rotation variants of *Turtle Barbarian* have about 80% higher peak memory consumption, but for our three movie scenes peak memory consumption is either much lower (*Llama*) or only slightly higher (*Barbershop* and *Train*). Whether we use more memory during STBVH build depends on the locations where the MBSAH chooses to perform temporal splits. Temporal splits performed at the top of the tree cause a higher overhead, and temporal splits performed more down in the tree cause a lower overhead. Thus when many objects can get separated spatially by the MBSAH before doing temporal splits (such as for the *Turtle Barbarian Crowd* scene) memory overhead can be very low. Note that we need to double the build primitive array during STBVH build only because we process both subtrees after a temporal split in parallel. One could get around this issue by sequentializing the BVH build at this point. While this sequentialization can easily get integrated into a binary BVH builder, it turned out to be difficult to integrate into our builder for 4 and 8-wide BVHs.

5 CONCLUSION AND FUTURE WORK

We presented the STBVH, a new approach for rendering multi-segment motion blur using a bounding volume hierarchy that stores spatial linear bounds and temporal bounds. We demonstrated that it performs competitively to building multiple, independent linear motion BVHs for a global number of time segments, but needs significantly less memory as it can spatially separate objects before introducing temporal separation, and it increases temporal resolution only where it is required. Thus the STBVH can store complex scenes with objects having widely varying time resolutions in an efficient, compact form, without sacrificing rendering performance.

We believe that the combination of spatial linear bounds, which tightly bound the typical case of linear motion, with temporal bounds, which allows the data structure to reduce the complexity of non-linear animations, makes this data structure a great candidate for adoption in movie production. Our algorithm can particularly play out its strengths for renderings where high-quality motion blur is desired for the main characters or objects, and few time steps are sufficient for other parts of the scene.

As future work, we would like to extend the STBVH and MBSAH to handle both animated and static geometries at the same time, so the entire scene could be stored in a single, unified BVH to improve ray traversal performance. We further would like to evaluate extensions to the MBSAH to reduce the data structure size even further, which could make our λ factor unnecessary.

The source code of our implementation of the STBVH approach can be found in the Embree Ray Tracing Kernels 2.16.0 [Woop et al. 2017].

6 ACKNOWLEDGMENTS

The *Llama* and *Train* scenes from the *Caminandes 3: Llamigos* movie (www.caminandes.com) are copyright (CC) by Blender Foundation. The *Barbershop* scene from the *Agent 327: Operation Barbershop* movie (agent327.com) is copyright (CC) by Blender Foundation based on original characters (C) by Martin Lodewijk.

The *Turtle Barbarian* model is courtesy of Autodesk and was originally modeled by Jesse Sandifer (sandpiper.artstation.com) for the Warriors Autodesk® Mudbox® Challenge.

REFERENCES

Jiří Bittner and Daniel Meister. 2015. T-SAH: Animation Optimized Bounding Volume Hierarchies. *Computer Graphics Forum (Proceedings of Eurographics 2015)* 34, 2 (2015), 527–536.

Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*. 1–6.

Per H. Christensen and Wojciech Jarosz. 2016. The Path to Path-Traced Movies. *Foundations and Trends in Computer Graphics and Vision* 10, 2 (Oct. 2016), 103–175.

Andrew S. Glassner. 1988. Spacetime Ray Tracing for Animation. *IEEE Computer Graphics and Applications* 8, 2 (March 1988), 60–70. https://doi.org/10.1109/38.504

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

Leonhard Grünschloß, Martin Stich, Sehera Nawaz, and Alexander Keller. 2011. MSBVH: An Efficient Acceleration Data Structure for Ray Traced Motion Blur. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. ACM, 65–70.

Johannes Hanika, Alexander Keller, and Hendrik P. A. Lensch. 2010. Two-Level Ray Tracing with Reordering for Highly Complex Scenes. In *Proceedings of Graphics Interface 2010*. 145–152.

Qiming Hou, Hao Qin, Wenyao Li, Baining Guo, and Kun Zhou. 2010. Micropolygon Ray Tracing with Defocus and Motion Blur. *ACM Transactions on Graphics* 29, 4 (2010).

Jens Olsson. 2007. *Ray-Tracing Time-Continuous Animations using 4D KD-Trees*. Ph.D. Dissertation. Lund University.

Matthew Quail. 1996. *Space Time Ray Tracing using Ray Classification*. Ph.D. Dissertation.

Ingo Wald, Solomon Boulos, and Peter Shirley. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics* 26, 1 (2007).

Ingo Wald and Vlastimil Havran. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 61–69.

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* 33, 4 (2014), 8.

Sven Woop, Carsten Benthin, and Attila T. Áfra. 2017. Embree Ray Tracing Kernels, https://embree.github.com. (2017).