Masterpraktikum aus Computergraphik und Digitaler Bildverarbeitung (SS2009)
Martin Kinkelin

# GPU Volume Raycasting using Bounding Interval Hierarchies

Martin Kinkelin (0326997)

**Abstract**

*Traditional Direct Volume Raycasting (DVR) on the GPU is limited to uniform voxel grids stored as 3D textures. This approach is not optimal for sparse data sets or data sets with highly varying frequencies because it requires a trade-off between data structure size and the maximum reproducible frequency and it lacks implicit empty space skipping during raycasting.*
*In this paper we present another approach, applying the Bounding Interval Hierarchy (BIH), a hierarchical spatial subdivision of elements traditionally used to accelerate surface raytracing, to volume raycasting on the GPU. Although connectivity information between voxels is lost and the texture filtering power of GPUs cannot be exploited, we show that it may be a viable alternative for DVR and that the approach is generic, allowing all sorts of renderable voxels (not-overlapping finite volume elements/primitives such as cuboids, ellipsoids and truncated Radial Basis Functions) for different tasks like rendering point sets as particle systems (e.g. using spherical voxels) and rendering volumes derived from traditional uniform grids (with implicit empty space skipping and the option for different levels of detail).*

## 1. Introduction

Most volume visualizations are currently based on Cartesian grids of voxels. A uniform grid is a very simple structure with some nice properties: trivial connectivity between voxels, no hierarchical overhead and last but not least hardware support for GPU-based Direct Volume Rendering (DVR). They are also very efficient for dense data sets with a small band of frequencies.

Uniform grids are not optimal for sparse data sets and large frequency bands though as the grid's resolution is a trade-off between memory footprint and the maximum reproducible frequency. Additionally, a hierarchical structure is often useful to derive different levels of detail.

Hierarchical structures to accelerate ray-casting have been extensively analyzed for surface ray-tracing applications, especially for partitioning triangular meshes. Currently, the *k*d-tree is widely considered to be the most efficient structure for ray-tracing static scenes on the CPU. Due to GPU limitations, Bounding Volume Hierarchies (BVHs) used to outperform *k*d-trees on the GPU though. We focus on a hybrid structure, combining the advantages of *k*d-trees and BVHs: the Bounding Interval Hierarchy (BIH). The structure and associated algorithms are covered in detail in section 3.

In section 4, we apply the BIH to volume ray-casting. Volume ray-casting poses some additional challenges compared to surface ray-casting, as most applications not only require the first intersection of the ray with an element, but up to all intersections with hit elements in correct order to allow for a composite view into the volume.

Some implementation details and optimizations are presented in section 5. In section 6, we evaluate the hierarchical BIH structure for different volumetric tasks and see how the traversal algorithm performs on current GPUs.

## 2. Related work

Wächter and Keller [WK06] introduced the Bounding Interval Hierarchy and compared it to a state-of-the-art *k*d-tree in a highly optimized CPU triangle ray-tracer. They conclude that the BIH is vastly more memory-efficient and is able to rival and even outperform the *k*d-tree in most test scenes. Additionally, the BIH's construction time is significantly better, making it suitable for dynamic scenes too.

A similar structure has been proposed by Eisemann et al. [EWM08]: the single slab hierarchy. It differs by allowing non-parallel bounding planes for both children so as to trim the child volumes more efficiently. Construction and traversal are more complicated compared to a BIH though, the latter preventing us from chosing it for a GPU renderer.

They compare the single slab hierarchy to the Bounding Volume Hierarchy (BVH) in a CPU-based triangle ray-tracer and conclude that their structure outperforms the BVH by a large margin both in memory footprint and achieved frame rates. They also list some other related structures.

Thrane and Simonsen [TS05] developed a GPU triangle ray-tracer and compared different acceleration structures: uniform grids, *k*d-trees and BVHs. They conclude that the BVH was the most memory-efficient structure and that it yielded the best frame rates too. Their implementation is now 4 years old; their results are based on a Geforce 6800 Ultra card. Current GPUs offer more flexibility (unified shader model 4.1 vs. 3.0) and vastly more performance, partly due to a unified shader architecture.

## 3. The Bounding Interval Hierarchy

The Bounding Interval Hierarchy (BIH) [WK06] is based on two different partitioning approaches: the *k*d-tree and the Bounding Volume Hierarchy (BVH). Although it is an object-partitioning scheme and therefore resembles more a BVH, the construction and traversal algorithms are very similar to those employed for *k*d-trees.

A *k*d-tree is a binary space partitioning (BSP) tree - each inner node in the hierarchy splits the space it contains into 2 disjoint halves and assigns them to its 2 children. The split is performed by a $(k-1)$-dimensional hyperplane (e.g. a line in 2D, a plane in 3D) perpendicular to one of the $k$ principal axes. Such an aligned splitting plane is compactly defined by an axis (e.g. $x$, $y$ or $z$ in 3D) defining the normal vector and an offset defining the distance from the origin. More importantly, the space contained by *k*d-tree nodes is enclosed by an axis-aligned box, a straightforward and intersection-friendly primitive.

The tricky part during the creation of the *k*d-tree is finding an optimal splitting plane. The Surface Area Heuristic (SAH) is widely used to construct efficient *k*d-trees for surface ray-tracing, e.g. to partition triangular meshes [WK06]. Ideally, both following criteria should be met:

- No elements intersect the splitting plane.
  Intersecting elements need either to be subdivided into subelements or to be assigned to both children and hence need to be further partitioned in both subtrees. This results in increased tree depth and size and likely in multiple, redundant intersection tests of an element with a ray.
- The probability of a random ray to hit elements in the left child equals that of hitting elements in the right child to optimize traversal.

The essential splitting part is handled differently by BIHs. Instead of specifying 1 splitting plane per inner node, an inner BIH node stores 2 planes parallel to the splitting plane: 1 bounding plane for each child, based on the axis-aligned bounding boxes (AABBs) of the children separated by the splitting plane (see figure 1). This means that empty space along the splitting axis inbetween the children is eliminated early and that the children may also overlap. Handling elements intersecting the splitting plane is hence very simple compared to *k*d-trees: each element is assigned to exactly 1 child, e.g. by classifying the center of an element's AABB into the positive or negative half-space defined by the splitting plane. For these reasons, a much simpler heuristic for the optimal splitting plane may be used, e.g. an intuitive one independent from the contained elements and based solely upon trying to keep a node's shape as compact (cubic) as possible, e.g. by splitting in the middle of a node's longest axis.

### 3.1. Construction

1. Calculate the axis-aligned bounding box (AABB) of all elements and use it as the box enclosing the root node.
2. Does the current node (enclosing box + contained elements) fulfill the leaf criteria? Then store a leaf and stop the current recursion, otherwise continue with the next step.
3. Select a splitting plane (axis and offset) for the inner node.
4. Assign each contained element to exactly 1 child (e.g. to the left child if the AABB center is on or on the left side of the splitting plane, to the right child if on the right side), generating 2 disjoint lists and calculating the corresponding bounding planes at the same time.
5. Store both bounding planes in the inner node as well as the pointer(s) to the children.
6. Process the children recursively (back to step 2 with the corresponding enclosing boxes and element lists).

The construction of a BIH is similar to that of *k*d-trees. Elements intersecting the splitting plane do not need to be treated differently though; additionally, the construction is augmented by the partial computation of the children's AABBs (maximum extents for the left child, minimum extents for the right child, along the parent's splitting axis). The idea is borrowed from Bounding Volume Hierarchies (BVHs), an object-partitioning scheme where each node stores the complete bounding volume of its contained elements. Due to partitioning of the elements being analog to the quicksort algorithm (pivot element = splitting plane offset, partitioning into left and right elements), its average time complexity is $O(N \log N)$, $N$ being the total number of elements. The bounding planes of the children are computed as part of partitioning the elements by simply storing the maximum coordinate (along the splitting axis) of all left elements' AABBs and the minimum coordinate of all right elements' AABBs.

Because the boxes enclosing BIH nodes are only trimmed on 1 side per tree level instead of all 6 sides per BVH node, BIH nodes may contain empty border spaces. Hence one of the children may be empty, containing no elements if the

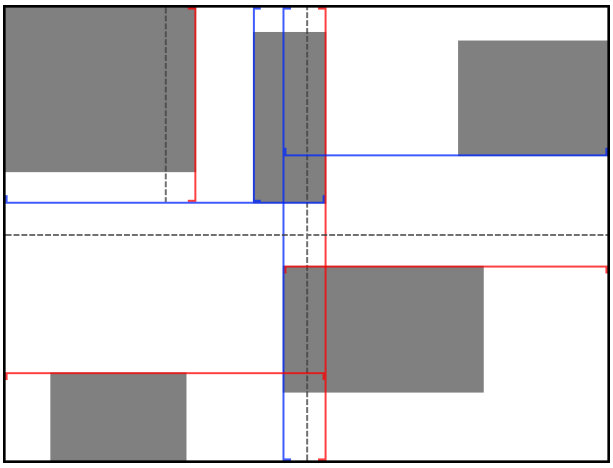

**Figure 1:** *Partitioning 5 elements (illustrated by their light-grey AABBs) in 2D using a BIH by splitting in the middle of a node's longest axis and stopping as soon as a node contains a single element. The splitting planes are depicted as dark-grey dashed lines, the bounding plane of a node's left child as a red line and that of the right child as a blue line. The root node's splitting plane is the central vertical one, assigning 3 elements to the left child and 2 elements to the right one; the children overlap. The left child is partitioned into the bottom-left leaf and another inner node separating both top-left leafs. The right child's children are both leafs.*

splitting heuristic does not directly depend upon the contained elements or if a node containing a single element is further split to trim its enclosing box. The intuitive solution for empty children is to store invalid bounding planes (e.g. $\pm$inf, resulting in non-intersectable children) in the parent node and not to store empty leaf nodes altogether because they are never going to be accessed during traversal. A more elaborate solution consists in using both planes of an inner node as the bounding planes of the single non-empty child along the splitting axis, trimming the child's enclosing box on 2 sides per tree level. This may save inner nodes and hence reduce tree depth and size, at the cost of having to handle this additional special case when constructing and traversing the tree.

### 3.2. Ordered traversal

We focus on the ordered traversal as it allows for early ray termination and ordered leaf intersections. The ordered traversal of a BIH, depicted in algorithm 1, is very similar to that of a *k*d-tree. The only differences are:

- the necessity to intersect the ray with 2 planes instead of 1 per hit inner node,
- an additional intersection case (no child hit in case the ray only intersects the empty space inbetween the children), and
- the possibility that elements contained by the second hit child intersect a ray before elements contained in the first hit child. This can only occur in space shared by both children and thus only if children overlap.

**Algorithm 1** Traversing the BIH tree for a given ray.

```
1:  push the root node onto the empty stack
2:  while there are nodes on the stack, i.e. subtrees to be traversed do
3:      pop the topmost node from the stack, representing the root of the deepest subtree
4:      while the node is an inner node do
5:          intersect the ray with the children
6:          if no child is hit then
7:              continue the outer loop with the next subtree
8:          else if only one child is hit then
9:              descend to that child
10:         else
11:             push the second hit child onto the stack
12:             descend to the first hit child
13:         end if
14:     end while
15:     render the leaf
16: end while
```

The complete intersection algorithm is illustrated in algorithm 2. BIH construction and traversal are numerically stable due to the comparison and min/max operations used when partitioning the elements and calculating the bounding planes (and hence the enclosing boxes too) as well as the $(t_{in}, t_{out})$ intervals. The intersection algorithm relies on the IEEE 754 floating-point standard because rays perpendicular to the splitting axis cause a division by 0, which is defined as inf$\cdot$ sign(dividend) in the aforementioned standard.

**Algorithm 2** Intersecting the children of an inner BIH node with a given ray $\vec{r}(t) = \vec{o} + t\vec{d}$. The node's axis-aligned *Box* (consisting in a *Min* and a *Max* corner) as well as the entry and exit distances of the ray ($t_{in}$ and $t_{out}$) are known (from the top-down traversal).

1: // compute the boxes enclosing the children's volumes
2: // *axis* is the index of the node's splitting axis
3: $Box_0 \leftarrow Box; Box_0.Max[axis] \leftarrow planeOffsets_0$
4: $Box_1 \leftarrow Box; Box_1.Min[axis] \leftarrow planeOffsets_1$
5:
6: // intersect the children's bounding planes with the ray
7: $t_0 \leftarrow \frac{planeOffsets_0 - \vec{o}[axis]}{\vec{d}[axis]}$
8: $t_1 \leftarrow \frac{planeOffsets_1 - \vec{o}[axis]}{\vec{d}[axis]}$
9:
10: // determine the traversal order depending upon $\vec{d}$
11: $firstIndex \leftarrow \vec{d}[axis] \geq 0 \; ? \; 0 : 1$
12: $secondIndex \leftarrow 1 - firstIndex$
13:
14: /* compare the first child's intersection to the entry point of the parent */
15: **if** $t_{firstIndex} > t_{in}$ **then**
16: the first child enclosed by $Box_{firstIndex}$ is hit
17: its exit distance is $\min(t_{out}, t_{firstIndex})$, the entry distance is still $t_{in}$
18: **end if**
19:
20: /* compare the second child's intersection to the exit point of the parent */
21: **if** $t_{secondIndex} < t_{out}$ **then**
22: the second child enclosed by $Box_{secondIndex}$ is hit
23: its entry distance is $\max(t_{in}, t_{secondIndex})$, the exit distance is still $t_{out}$
24: **end if**

Algorithm 1 uses a stack to store the root nodes of the subtrees which are hit by the ray but need yet to be traversed (immediately after the first hit sibling's subtree has been traversed). The entry and exit distances $t_{in}$ and $t_{out}$ need to be stored for each root to be able to continue intersection testing from that node. If leaf rendering depends upon a leaf's enclosing box, the roots' boxes need to be stored on the stack too, although in that case the $(t_{in}, t_{out})$ interval may be derived by a full ray-AABB intersection test. To cover the worst case, the stack needs to be able to hold a number of elements that equals the maximum depth of the tree. Although stackless solutions exist (at least for *k*d-trees: *k*d-restart and *k*d-backtrack [HSHH07]), these approaches are work-arounds for temporary GPU limitations. If the stack itself (i.e. a local array for every ray) is not a bottleneck, employing a stack is the most elegant and theoretically most efficient method for tree traversal as both other approaches result in high additional traversal cost, and *k*d-backtrack additionally in increased memory footprint.

### 3.3. Memory layout

Inner BIH nodes are defined by a splitting axis, the offsets of both children's bounding planes and the pointer(s) to the children. By using 16-bit fixed-point values for the plane offsets, 2 bits for the node type (3 splitting axes + leaf node) and 30 bits for a single pointer/index, an inner node requires only 8 bytes. A single pointer to one child suffices in case the pointer to the other is implicitly derivable, e.g. by storing the tree in pre-order layout (depth first: root, left subtree, right subtree), where the left child is stored right after the parent and only a pointer to the right child is required.

The definition of a BIH leaf node is flexible and depends upon the application. All parent nodes' bounding planes define a leaf's enclosing box, therefore an axis-aligned box is implicitly given. The node type is a common field for all nodes, hence 2 bits are reserved. If leafs contain multiple elements, a pointer to a elements list may suffice. On the other hand, elements may be stored directly in the BIH tree if the elements are fully partitioned so that every leaf contains a single element. If nodes containing a single element are further split (generating empty children) until a leaf's enclosing box is trimmed on all 6 sides, the element's AABB equals the leaf's implicit enclosing box and does therefore not need to be stored. In our example, there are 62 bits available to store an element's additional attributes if the size of a leaf node should equal the size of an inner node. The 62 bits may be subdivided in $3 \cdot 16 + 14$ bits to store 4 scalar attributes, e.g. color and opacity or a scalar density and an associated 3D gradient (see table 1).

In a perfect binary tree, where every inner node has 2 (non-empty) children, the total number of nodes is $2 \cdot$ #leafs $-1$. As discussed earlier, BIH leafs may be empty though, so the total number of nodes cannot be derived a priori from the number of non-empty leafs. The stated formula resolves to the lower bound for the total number of BIH nodes, every empty node increases the number by 1 (if empty nodes are not stored). The number of empty nodes depends upon the splitting heuristic and the data set; some examples are presented later on.

### 3.4. Comparison between *k*d-tree, BIH and BVH

The main difference between a BIH and a binary BVH (of AABBs, split by aligned planes) is the node definition. A BIH node's enclosing box is trimmed on 1 side per tree level and therefore requires 1 plane offset to be stored. A BVH does not exploit likely coherence of the other 5 bounding planes between the AABBs of parents and children: a BVH node stores all 6 plane offsets of its contained elements' AABB. During traversal, each node is independently intersected with the ray by a full ray-AABB intersection test.

BVH nodes are usually at least twice as large as BIH nodes. E.g. when using $6 \cdot 16$-bit fixed-point values for the AABB coordinates and 2 bits for the node type, there are

| **Inner BIH node** | left plane offset | right plane offset | index | | splitting axis index $\in \{0,1,2\}$ |
|---|---|---|---|---|---|
| | 16 bits | 16 bits | 16 bits | 14 bits | 2 bits (node type) |
| **BIH leaf node** | 1st attribute | 2nd attribute | 3rd attribute | 4th attribute | 3 (leaf node) |

**Table 1:** *Memory layout of BIH nodes. Each node consists in 4 · 16 bits = 8 bytes.*

| | ***k*d-tree** | **BIH** | **Binary BVH** |
|---|---|---|---|
| Typical node size | 8 bytes | 8 bytes | at least 16 bytes |
| Typical tree depth | rather high: elements intersecting a splitting plane are duplicated, potentially lots of empty leafs if empty space is eliminated | rather low: some empty leafs | low: no empty leafs |
| Construction | costly, as the efficiency greatly depends upon an involved splitting heuristic | cheap, as simple splitting heuristics already deliver satisfying results; a BIH splitting heuristic may be even simpler than a BVH's (by not computing the full AABB but using only the enclosing box), at the cost of potentially more empty leafs | |
| Intersection/traversal | fast, both children are intersected at once per inner node by intersecting a single plane | minimally slower than the *k*d-tree (2 planes per inner node, additional intersection case), but flatter tree | slower due to full ray-AABB intersection test per node; independent from the parent (no enclosing box and/or $(t_{in}, t_{out})$ tracking) |
| Leafs containing a single element and implicitly representing its AABB by its enclosing box | not possible unless no elements intersect any splitting plane at all; even so, many empty leafs may be needed to trim the enclosing box | trivial: partition until a node's enclosing box equals the single contained element's AABB, at the likely cost of a deeper tree | trivial: partition until a node contains a single element |

**Table 2:** *Comparison between kd-tree, BIH and BVH*

30 bits left if using 16 bytes per node. The 30 bits may be used for either a pointer (inner nodes) or leaf attributes (while our proposed BIH implementation allows 62 bits for the attributes). Splitting a BVH node always results in 2 non-empty children, so the number of BVH nodes only depends upon the number of leafs and the hierarchy may be flatter than an equivalent BIH. It must be noted though that the number of empty BIH leafs can be reduced by employing a BVH splitting heuristic, e.g. by computing the full AABB of the contained elements (instead of a single bounding plane) and splitting in the middle of its longest axis. In this case, empty leafs are only generated when trimming nodes containing a single element. The BVH splitting heuristic however may cause very unbalanced trees because the volumes of two children can differ greatly.

The main difference between *k*d-trees and the other 2 hierarchical structures is that *k*d-trees partition space, not elements. An involved splitting heuristic is of key importance for good efficiency; the longer construction times still do not prevent the *k*d-tree from consisting in more levels and nodes than BVHs and BIHs. Additionally, empty leafs cannot be simply omitted in the memory layout of the *k*d-tree unless using tricks to indicate empty children in the parent node and watching out for them during traversal. A general comparison of the 3 hierarchical structures is presented in table 2.

## 4. Application to volume ray casting

Most volume visualizations rely on ordered intersections because most applications make extensive use of alpha blending to allow for a composite rendering of the volume. The blended color contributions need to be gathered either in back-to-front or front-to-back order, the latter allowing to terminate a ray as soon as the overall opacity exceeds a certain threshold (early ray termination). Rasterization-based approaches are not suited for complex semi-transparent tasks because the rasterized primitives need to be sorted by their distance from the view-point beforehand while ray casting offers implicit sorting.

We briefly mentioned that elements contained by the second hit child may intersect a ray earlier than elements in the first hit child; this is a common issue of all object-partitioning approaches. Figure 2 illustrates an example of such cases which can occur in the space shared by both chil-

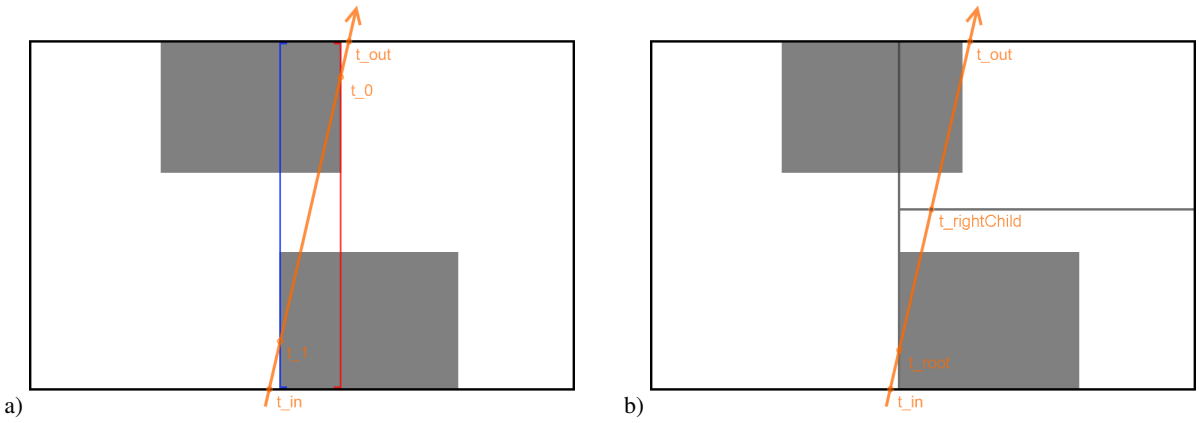


**Figure 2:** *Examples of disordered intersections when traversing a BIH (a) and a kd-tree (b).*
*a) Due to the positive ray direction along the splitting axis (x), the left leaf is traversed before the right one. The left leaf contains the upper element, the right child the lower one, hence the element intersections are not processed in the correct order.*
*b) An analog problem may occur when using kd-trees and assigning elements intersecting the splitting plane simply to both children. The left leaf contains the upper element, the right child both elements. The right child is further partitioned into the bottom-right and top-right leafs. The traversal order is as follows: left leaf (top element), bottom-right leaf (bottom element), top-right leaf (top element again).*

dren. It also shows that this problem concerns *k*d-trees too, as long as elements intersecting the splitting plane are not subdivided but assigned to both children. There is a solution for *k*d-trees though: only element intersections in the leaf's $(t_{in}, t_{out})$ interval, i.e. inside the leaf should be considered valid so as to simulate a subdivision of elements.

In surface ray-tracing scenarios, only the first intersection of the ray with an element is required. In this case, after finding an intersection, the subtrees stored on the stack need to be checked for earlier intersections as long as there are nodes which are hit before the first intersection so far. So the potentially disordered intersections in the overlapping space of 2 children do not represent a serious issue. They pose a problem for most volume visualizations though unless ordered traversal is not required. Possibilities to circumvent this issue include:

- No overlapping children: this condition is satisfied if the elements are positioned on a Cartesian grid and the distance between 2 neighboring grid points is at least as long as the maximum size of all elements.
- Ignore: allow potentially disordered intersections in overlapping space if it is rather small and artifacts are therefore barely noticeable.
- Construct a *k*d-ish BIH: either by splitting elements into sub-elements or by setting both bounding planes to the splitting plane and assigning overlapping elements to both children. In the latter case, the boxes enclosing the BIH nodes are no longer guaranteed to contain the contained elements' AABB. Obviously a more elaborate splitting heuristic needs to be employed to minimize the number of overlapping elements, analog to *k*d-tree construction. Such a *k*d-ish BIH still provides the advantage of eliminating empty space and of being able to omit empty leaf nodes altogether.

By traversing a BIH for a given ray, all leaf nodes pierced by the ray are visited in a specific order (usually front-to-back to exploit early ray termination, with the just mentioned possibility of slight disorders). In volume visualization applications, the leafs are rendered in some way. Each leaf needs to be renderable independently as there is no real connectivity between leafs - a leaf and its nearest neighboring leaf may already be separated by the root node's splitting axis. The only form of potentially useful connectivity is the ordered traversal of the leafs, i.e. a connection between a hit leaf and the previously hit one. So each leaf needs to contain volumetric primitives which are intersected with the ray when the leaf is rendered.

### 4.1. Empty space skipping for Cartesian grids

In traditional Direct Volume Rendering (DVR) on the GPU, a 3D texture representing a Cartesian grid of voxels is resampled along an intersecting ray segment in steps of fixed length. If the volume is sparse, there are lots of transparent voxels and hence many unnecessary samplings are performed. Octree- and BVH-based empty space leaping techniques have been proposed to skip samples in transparent regions (empty nodes), but we do not know of any GPU-based implementation.

BIHs are also suited for this task because empty space is implicitly eliminated. A coarse BIH may be used to identify non-empty cells in the Cartesian grid. A cell pierced by a

ray is then regularly resampled. The BIH leafs do not need to map 1:1 to cells in the grid; a leaf's box can be mapped to an arbitrary cell in the grid by specifying only a 3D offset and the extents. Therefore different regions of the volume could use different resolutions, based on the frequencies in the regions.

Inner BIH nodes could also store some additional attributes like the minimum and maximum density of all contained voxels. This may be useful for rendering iso-surfaces as large regions may be skipped if the requested iso-value is outside a node's density interval.

### 4.2. Semi-transparent particle systems

Particle systems are often rendered by exploiting the GPU's point-sprites rendering or geometry shader capabilities: the center point is expanded to a billboard, texturized and rasterized. The restriction of this rasterization approach is that the particles should be opaque, otherwise the points need to be sorted by their depth from the current view-point to allow for correct blending.

Ray-casting techniques offer implicit sorting and are therefore well suited for semi-transparent particle systems too. [KAH07] propose to treat each point as a spherical particle with a specific radius. The weight of a particle's color contribution to a ray depends upon the absolute distance $d$ of the point from the ray, relative to its radius $r$:

$$\alpha(d,r) = \begin{cases} \exp(-\kappa(\frac{d}{r})^2), & \text{if } d \leq r \\ 0, & \text{if } d > r \end{cases}$$

A set of spheres is easily subdividable into a BIH. Each leaf may represent one sphere so that its implicit enclosing box defines its position and its size. The remaining 62 bits may be used for additional attributes like color and opacity. When a ray hits a leaf, the distance of the center from the ray is computed to obtain the weight of the color contribution. [KAH07] shows how particle systems can be combined with structured grids, e.g. for cosmological simulations where particles represent galaxies, stars or planets and the grid encodes gas, fog etc.

### 4.3. Completely BIH-based DVR

A BIH is also able to represent a discretized, continuous volume. For example, a BIH may be based on the non-transparent cuboid voxels of a Cartesian grid, e.g. by embedding a single voxel into every BIH leaf. This approach offers some advantages over traditional DVR:

- Required memory for the volume may be reduced significantly if there are lots of transparent voxels.
- Transparent regions of the volume are not taken into account during rendering.
- All voxels hit by a ray may contribute to the final pixel color. In traditional DVR, the precision depends upon the length of a sample step.
- The hierarchical structure allows to easily derive different levels of detail.

The BIH has one big disadvantage over a uniform grid though: we lose the connectivity between neighboring voxels, i.e. there is no trivial/fast way to find a voxel's neighbors. This leads to 2 disadvantages compared to traditional DVR:

- There is no hardware filter support, i.e. voxel attributes cannot be interpolated trilinearly.
- Gradients cannot be computed on-the-fly using the central difference algorithm.

There is at least a partial solution for the first issue. Firstly, the voxel gradients can be precomputed, e.g. by using the simple central difference. The gradient can then be embedded in every BIH leaf. This does not allow for smooth surface shading, but the gradient $\vec{g}$ can also be used to trilinearly approximate the density $d$ at a point $\vec{x}$ inside a leaf with the central density $d_{center}$, the center point $\vec{c}$ and the size $\vec{s}$:

$$\vec{o}(\vec{x}) = \frac{\vec{x}-\vec{c}}{\vec{s}} \ldots \text{offset of } \vec{x} \text{ from } \vec{c} \text{ in } [-0.5, 0.5]^3$$

$$d(\vec{x}) = d_{center} + \vec{g} \cdot \vec{o}(\vec{x})$$

Rendering the leafs can be accomplished in multiple ways:

- Multi-sample BIH-DVR: sampling the ray segment in equidistant steps, just like traditional DVR but on a per-leaf basis. Care needs to be taken to ensure equidistant steps between leafs consecutively hit by a ray, i.e. starting to sample at $t_{in}$ for every leaf is not going to work; sampling may start at multiples of the sample step length, for example. If a ray segment inside a leaf is shorter than the sample step length, the leaf may not contribute to the final pixel value at all, analog to traditional DVR.
- Computing the densities at $t_{in}$ and $t_{out}$ as well as the ray segment length and then using a pre-integrated transfer function [EKE01] (3D due to the variable ray segment length) to look up the overall color contribution.
- Single-sample BIH-DVR: Computing the average density along the ray segment and using it as constant density along the whole segment. As the density is approximated trilinearly, the average density is the density in the middle of the segment, so a single sample per leaf is enough. To account for the variable segment length $l$ (relative to the sample step length), the color contribution (with base opacity $\alpha \in [0,1]$) is weighted by the opacity correction formula:

$$\alpha'(l,\alpha) = 1 - (1-\alpha)^l$$

To yield plausible results, the transfer function (TF) should not include high frequencies and it should be approximately piece-wise linear (especially along a ray segment inside a leaf), so that the average density maps approximately to the average TF classification (i.e. color and base opacity) along the segment.

## 5. Implementation details

Our approach consists in 2 stages:

1. Preprocessing: constructing the BIH and deriving levels of detail (LoDs) from it
2. Rendering: ray-casting on the GPU by traversing the BIH LoD and rendering the leafs

### 5.1. BIH

After computing the AABB of all elements, the elements are recursively partitioned into a BIH. The splitting plane is placed in the middle of a node's longest axis, trying to keep the node shapes as compact as possible. The bounding planes are calculated while assigning the elements to the children, saving redundant memory accesses compared to a two-pass approach. The assignment is accomplished by in-place partitioning (i.e. operating on the original list), swapping pairs of elements until the left and right ones are separated. The recursion is stopped if a node's enclosing box equals the single contained element's AABB.

To provide enough precision, the 16 bits per plane offset are used as fixed point values, 0x4000 representing 0 and 0xC000 representing 1 in normalized volume-space $[0,1]^3$. Precision is reduced to 15 bits (32768 uniform steps per axis) to represent $\pm$inf as 0x0 (-0.5 in normalized volume-space) and 0xFFFF (1.5) for the bounding planes of empty children. This allows to offset the bounding planes on-the-fly in the shader program, e.g. to adjust the size of spherical particles (and hence all nodes) interactively. The additional range results in the bounding planes of empty children remaining invalid (outside the volume's AABB) as long as the offset value is $< 0.25$ in normalized volume-space.

### 5.2. BIH Level of Detail (LoD)

One advantage of the hierarchical BIH structure is the ability to easily derive different levels of detail, e.g. by limiting the tree depth or the minimum node size. It also allows for compression by merging subtrees under certain conditions, e.g. if a node contains a dense set of elements with similar attributes to compress larger solid volumes.

To exploit these options, we calculate some statistical characteristics for each BIH node, including the attribute means and associated variances as well as the density of contained elements. These characteristics are computable in a bottom-up approach (i.e. the parent node combines the characteristics of its children to compute his own), thereby not affecting performance severely during tree construction.

The statistical characteristics associated with each node are evaluated when selecting a static level of detail to be rendered. A static LoD is a trimmed-down version of the BIH specialized for rendering on the GPU. During creation of the BIH LoD, each BIH node is checked against the LoD's leaf criteria:

- maximum depth in the tree,
- minimum element density $\frac{\sum V_{element}}{V_{box}}$, and
- maximum standard deviation of the contained elements' normalized density.

If a BIH node fulfills the leaf criteria, it is used as LoD leaf node, discarding its subtrees. Figure 3 demonstrates the impact of the 2 latter LoD parameters on the Vertebra data set.

LoDs might also be derived dynamically in the shader program, possibly using an adapative method, e.g. by limiting the descend depth with increasing pixel opacity because further color contributions need not be as precise as the results of the first hits.

### 5.3. Rendering

To allow for high performance, several aspects of current GPU architectures need to be regarded. Firstly, GPUs offer enormous computing power for parallelizable tasks like raytracing compared to generic CPUs. This is achieved by executing the same instructions on a large group of stream processors. The instruction set is a lot smaller than those of CPUs but equipped with some specialized 3D instructions. The first bottleneck arises with branching (if, while etc.) because the simultaneous threads may differ in the branch being taken. Currently there are two approaches to solve this problem: predicated branching, which consists in all threads evaluating all branches and hence useful for small branches, and dynamic branching, which allows to skip the wrong branch if the simultaneous threads are all coherent, i.e. take the same branch.

Another limitation of current GPUs are indexable temporary registers per thread. The number of these registers, usable for local arrays, is very low, especially for lower-end graphic cards. We need them for the per-thread stack, which in turn needs to be as large as the maximum tree depth to cover the worst case. If the available registers are too few, main memory is used (so-called scratch registers), hurting performance extremely. In any case, the stack occupies many registers and therefore the number of parallel threads on the GPU is decreased (due to a limited number of registers), which results in decreasing the GPU's ability to hide latencies.

We decided to use the programmable 3D pipeline (Direct3D 10) instead of taking a GPGPU approach, mainly due to the OpenCL standard not being supported by both NVIDIA (CUDA) and ATI (Stream) yet. Using the traditional pipeline allows to render directly to the framebuffer and to process only pixels actually intersecting the volume. A GPGPU approach might offer the flexibility to place the per-thread stack into shared memory, which is supposed to be only slightly slower than registers. That would reduce the number of required registers per thread significantly, so that more threads may be managed simultaneously and memory

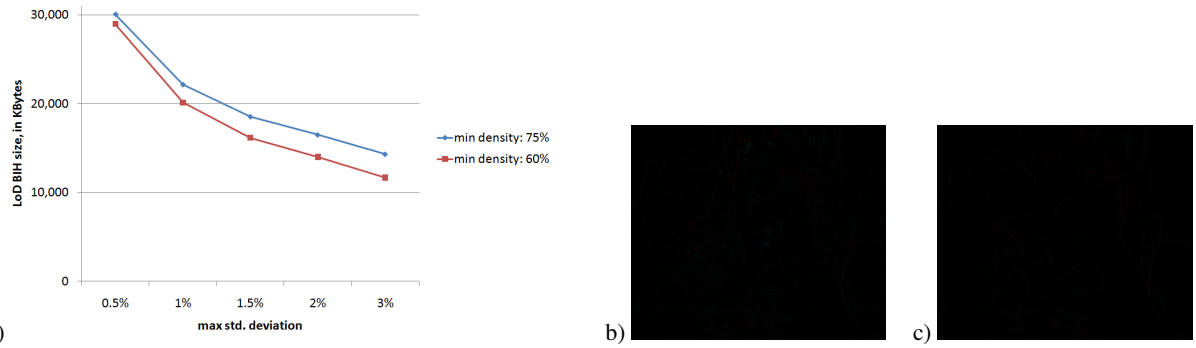


**Figure 3:** *Impact of 2 LoD parameters on the Vertebra data set, using all voxels with normalized density above 0.125 as BIH elements. The first parameter, the minimum density, controls how much empty space is tolerated inside an inner BIH node to be considered as LoD leaf candidate. If the standard deviation of the contained voxels' values is additionally less than the maximum standard deviation, the second crucial parameter, the inner BIH node is used as LoD leaf node, pruning its subtrees. a) Graph illustrating the impact on the LoD size using various combinations of the two parameters. b) Difference of the DVR reference frame and the BIH-DVR frame using a min density of 60% and a max std. deviation of 1.5%. c) Difference of the reference and the default LoD configuration: min density of 75%, max std. deviation of 1%.*

latency may be hidden more efficiently. On current ATI hardware, each thread may write to a maximum of $4 \cdot 16$ bytes, restricting the stack size extremely, so this is currently not an option.

To rasterize only pixels intersecting the volume covered by the BIH, the volume's axis-aligned bounding box is rendered as proxy geometry (using front-face culling). The vertex shader simply transforms the vertex position into clip-space and computes the vector from the eye-point to the vertex in object-space. This vector is then interpolated across the fragments. The pixel shader normalizes it to obtain the ray direction and transforms it to normalized volume-space $[0,1]^3$. After precomputing some often used values (reciprocals etc.), the BIH is traversed for the ray and the color contributions of hit leaf nodes are accumulated. The ray is terminated early once the accumulated opacity reaches a certain threshold near 1. To be able to use bit-wise operations and to rely on IEEE 754 floating point behavior when dividing by zero, the GPU needs to support shader model 4.0, supported by all DirectX 10 compatible graphic chips.

The traversal algorithm 1 for the GPU has been slightly modified to replace both loops by a single one. This increases performance due to the branching limitations mentioned previously. The stack size is adapted for the tree (i.e. set to the max tree depth). The roots on the stack are compactly defined by their enclosing boxes ($6 \cdot 16$ bits packed into a uint3) and their node index (another uint, therefore an element fits nicely into a uint4, i.e. exactly one register).

### 5.4. Memory layout of the BIH tree

The memory layout of the BIH structure to be used by the GPU needs to be chosen carefully. Firstly, GPUs support less memory than CPUs and required memory bandwidth should be minimized. Therefore the structure should be as compact as possible. Additionally, it should be usable as a texture to use it in the graphics pipeline. This requires all fields of a tree node to be of the same data type (integer/floating point, 16 or 32 bits) and poses restrictions on the number of fields (e.g. 3 for an RGB texture, 4 for an RGBA texture, or multiples of these values when using more than one texel per tree node).

We use a 4-component (RGBA) texture buffer and 16 bits (interpreted as unsigned integers) per component, resulting in 64 bits per texel. The nodes defined in table 1 therefore fit perfectly into a single texel. A texture buffer is a (large) read-only 1D array of texels, comparable to a 1D texture without severe size limitations (up to $2^{27}$ texels vs $2^{14}$ for 1D textures in Direct3D 10) and no filtering and floating point indexing options. Using a texture buffer instead of a 2D texture eliminates the overhead of converting indices to 2D coordinates and allows for a sequential caching behavior.

Traversing a BIH is a memory-intensive task. Exploiting caches is a necessity to hide memory latencies, therefore one should try to keep the children close to the parent. The pre-order tree layout (root, left subtree, right subtree) allows sequential memory access, but only if the left child is always the first one to be traversed. This constraint is only satisfied if all ray direction components are positive and is therefore inherently view-dependent.

Another possibility is to store both children of a node sequentially as a pair, so that the parent points to the left child and the right child's index is simply the next one (assuming the same node size for inner nodes and leaf nodes). We generalize this concept to tree chunks: the subtrees of a node pair are followed until some maximum relative depth, and

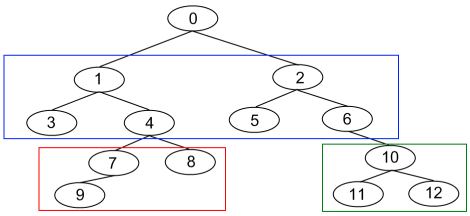


**Figure 4:** *Serialization of a tree into 3 tree chunks (blue, red and green) with a maximum tree chunk depth of 1. The number of each node represents its index in the final array. A tree chunk contains some levels of two subtrees and stores the contained nodes in breadth-first order. The child chunks are processed in depth-first order.*

the nodes are stored in breadth-first order. The rest of the subtrees are processed as child chunks in depth-first order (see figure 4). The advantage is that this way, the nodes in the next few levels should be already cached, independent from the traversal order. For example, if we use a chunk depth of 2, 3 levels of both subtrees (up to $2^{3+1}-2$ total nodes) are stored tightly as one chunk, a contiguous block in memory. The goal is to minimize latencies due to frequent jumping in the serialized binary tree and to rival octrees, whose inner nodes synthesize 3 levels of a binary tree.

## 6. Evaluation

### 6.1. BIH construction and storage efficiency

In the previous section, we mentioned that our implementation does not compute directly a BIH to be rendered. Instead, we compute an intermediate BIH, where all elements are partitioned until every leaf node's box equals the axis-aligned bounding box of the contained element. Additionally, every node is associated with statistical characteristics. This intermediate data structure is then used to quickly derive different static levels of detail for rendering. This is useful to find appropriate parameters and to experiment, but the intermediate BIH also requires a lot of storage, as our implementation of a full BIH node requires 40 bytes (compared to the 8 bytes of a LoD node). The intermediate BIH might also reach a high depth because nodes containing a single element are trimmed until the enclosing box equals the element's AABB (worst case: 5 additional levels).

Our primary data sets are popular uniform grids. The elements partitioned by the BIH are a subset of the grid's voxels: all voxels with a configurable minimum normalized density value (usually 0.1 to remove most of the noise). The gradients are obtained by central differencing.

The test system is a regular desktop PC equipped with a Phenom X4 9950 CPU @ 3 GHz, 4 GB of DDR2-1066 RAM and a single Radeon HD 4850 (GPU @ 725 MHz, 512 MB GDDR3 memory @ 1.1 GHz). The machine is running the Windows 7 RC and using the Catalyst v9.6 driver. The software is based on native C++ core algorithms and a .NET GUI.

Table 3 lists various measurements regarding BIH construction and memory footprint obtained from different uniform grid sources. Our first observation is that the BIH construction is quite quick considering the number of elements and that the default LoD's final footprint is usually significantly smaller than the uniform grid's. To be fair, not all source grids contain 16 bit values (some store the density in 8 bits), so the ratio depends upon the precision too. On the other hand, the LoD uses 14 bits for the density as well as $3 \times 16$ bits for the precomputed gradient for each element. The number of elements could be further reduced if the focus is on material borders - by not importing voxels with small gradient magnitude.

The memory reduction is high if:

- the volume is sparse and most voxels are considered noise (and hence not used as BIH elements), and/or if
- the volume contains regions of low frequency whereby a single LoD leaf node may represent a neighboring set of similar voxels. This applies especially to large grids.

Another observation is that the number of inner BIH nodes with a single child (i.e. a regular child and an empty leaf) is usually quite small, although it may get large (especially for non-uniformly positioned and sized elements). A large number results primarily from trimming a node containing a single element until its box equals the element's AABB. This trimming occurs in the deepest tree levels and hence the number of empty leafs may get large. This is mostly an issue for our implementation of the intermediate BIH (causing a large memory footprint) but not for the LoD: if the box enclosing a node with a single element is only slightly larger than the element's AABB, the node will be used as LoD leaf node (depending upon the LoD's element density parameter).

The BIH is a very compact data structure for general elements in space. Our implementation requires 64 bits per node; the number of inner nodes is usually only slightly larger than the number of leaf nodes, depending upon how exact the element AABB is to be approximated by the leaf boxes. If there is only a single empty leaf, the number of total nodes is 2 times the number of leaf nodes. In that case, one might say that an element consumes $2 \times 64 = 128$ bits in the BIH structure. 62 bits are available for element attributes. The remaining 66 bits are not simply overhead since the inner nodes define (or approximate) the element's AABB. A full AABB in 16 bits precision would require $2 \times 3 \times 16 = 96$ bits. So in case most of the 62 attribute bits can be exploited, the BIH's footprint may actually be smaller than a flat array of elements - while providing a hierarchical spatial subdivision.

| | | Lobster | Engine | Skull | Abdomen and pelvis | Head aneurysm |
|---|---|---|---|---|---|---|
| Grid | Dimensions | $301 \times 324 \times 56$ | $256 \times 256 \times 128$ | $256^3$ | $512 \times 512 \times 174$ | $512^3$ |
| | Size in KBytes (16 bit density) | 10,667 | 16,384 | 32,768 | 89,088 | 262,144 |
| Number of BIH elements (voxels with normalized density $\geq 0.1$) | | 341,256 (6.25%) | 1,352,439 (8.06%) | 2,058,103 (12.27%) | 16,183,116 (35.48%) | 16,481,795 (12.28%) |
| BIH | Construction time in ms | 702 | 2,777 | 4,633 | 35,646 | 40,435 |
| | Number of nodes | 726,844 | 2,780,802 | 4,680,234 | 32,833,739 | 39,012,660 |
| | % nodes with single child | 6.1% | 2.73% | 12.05% | 1.42% | 15.51% |
| | Max tree depth | 24 | 23 | 24 | 26 | 27 |
| | Size in KBytes | 28,393 | 108,626 | 182,822 | 1,282,568 | 1,523,933 |
| Default LoD | Construction time in ms | 78 | 281 | 546 | 858 | 1,982 |
| | Number of nodes | 515,349 | 1,367,240 | 3,318,436 | 5,063,523 | 11,931,875 |
| | Max tree depth | 24 | 23 | 24 | 26 | 27 |
| | Size in KBytes | 4,027 | 10,682 | 25,926 | 39,559 | 93,218 |
| | Size relative to uniform grid | 37.75% | 65.2% | 79.12% | 44.4% | 35.56% |

**Table 3:** *Various measurements regarding BIH construction and memory footprint for 5 uniform grid sources. The default LoD is based on the following criteria: a leaf node's voxel density is $\geq 75\%$ and the allowed standard deviation of the contained voxels' density is $\leq 1\%$; the maximum tree chunk depth is set to 2.*

### 6.2. BIH-based DVR performance and rendering quality

We implemented a flexible renderer, capable of using traditional DVR or BIH-based DVR methods. We focus on the comparison between traditional DVR and the single-sample BIH-based DVR described in subsection 4.3. Multi-sample BIH-based DVR is obviously slower than the single-sample version and does usually not enhance visual quality, but rather introduces additional ring artifacts just like traditional DVR in regions where the sampling step is too small for high-freqency regions. Although these artifacts are not produced by single-sample BIH-DVR (there is no undersampling as the ray segment's central sample is considered as uniform sample along the whole segment inside the leaf node), visual quality is degraded compared to traditional DVR because true trilinear filtering is only approximated by a single gradient for each leaf node. Pleasing results are obtained if either the gradient is small and hence if the grid's frequency is low, or if the opacity of each leaf's color contribution is rather low and artifacts are hidden.

The visual quality of BIH-based DVR therefore primarily depends upon the frequency of the uniform grid as well as the employed transfer function. The performance is affected primarily by the sparseness of the volume and the grid's frequency (regions of low freqency are compressed by larger leaf nodes). The transfer function also has a strong impact on the performance of both traditional DVR and BIH-DVR as high opacities lead to early ray termination.

Before moving on to the first comparisons, we need to describe our traditional DVR implementation. Traditional DVR is quite straight-forward: a ray segment inside the volume is resampled at equidistant steps. Our sampling step size relates to 1,024 steps along the longest axis of the volume (in object-space) to reduce the ring artifacts. Zero-density samples are skipped to increase performance. If the gradient is required, it is computed on-the-fly by sampling the 6 direct neighbours of each sample - using a filtered precomputed gradient would increase the grid's memory footprint by a factor of 4. The transfer function is a 1D RGBA texture, 256 texels wide and using 8 bits per component. Gamma-correction is applied (automatically by Direct3D 10 if using appropriate texture formats).

The performance of our single-sample BIH-DVR approach does not depend upon the sampling step size (the accuracy is fixed by the leaf nodes, there is no under- or oversampling) - it only affects the weight of each leaf's color contribution. The precomputed 16-bit gradient can be stored for free in the leafs, together with the 14-bit density, so the performance is not affected in case the gradient is required (for shading and/or assigning material borders a greater weight). In all tests, we used the default LoD parameters described in the previous subsection. The frame rates (fps) are included in the screenshots and obtained from the Fraps utility. The viewport resolution is 760x640 pixels (almost half a million pixels, i.e. almost twice as many pixels as the popular 512x512 viewport).

#### 6.2.1. The engine data set

Our first test data set is the popular Engine in $256^3$ resolution. It is not extremely sparse if discarding voxels whose normalized density is below 10% and is composed of many round shapes. These provide a challenge for the BIH-DVR approach as it is based on rendering the cuboid leaf nodes (single voxels or dense blocks of voxels with similar density) independently from each other. The volume is quite noise-free and therefore suitable for nice renderings.

Looking at the first results, it became pretty obvious that the frequency of the data is too high - the density approximation using the gradient is not precise enough to compete visually with traditional DVR unless using very low opacities. Therefore we use an additional BIH for comparison, based on a supersampled version of the source grid. For this data set, we selected a resampling factor of 1.5 for every dimension, resulting in $1.25^3 = 3.375$ as many voxels and potential BIH elements. This obviously increases BIH construction time and tree size. To be able to compare the visual results, the precomputed gradients also need to be scaled by the same factor as they are based on a nearer neighbourhood.

| | BIH-DVR 1 | BIH-DVR 1.5 | DVR |
|---|---|---|---|
| Resampling factor | 1 | 1.5 | 1 |
| BIH construction | 2,762 ms | 10,218 ms | - |
| Max LoD depth | 23 | 25 | - |
| Data structure size | 10,682 KB | 32,769 KB | 32,768 KB |

**Table 4:** *Characteristics of the 3 sources for the engine renderings. Please note that the nearly identical size of the BIH-DVR 1.5 LoD and the original uniform grid is pure coincidence. This demonstrates the storage efficiency of a BIH: it is based on 3.375 times as many voxels as the original grid and additionally contains a precomputed gradient per leaf node, which would increase the grid's size by a factor of 4.*

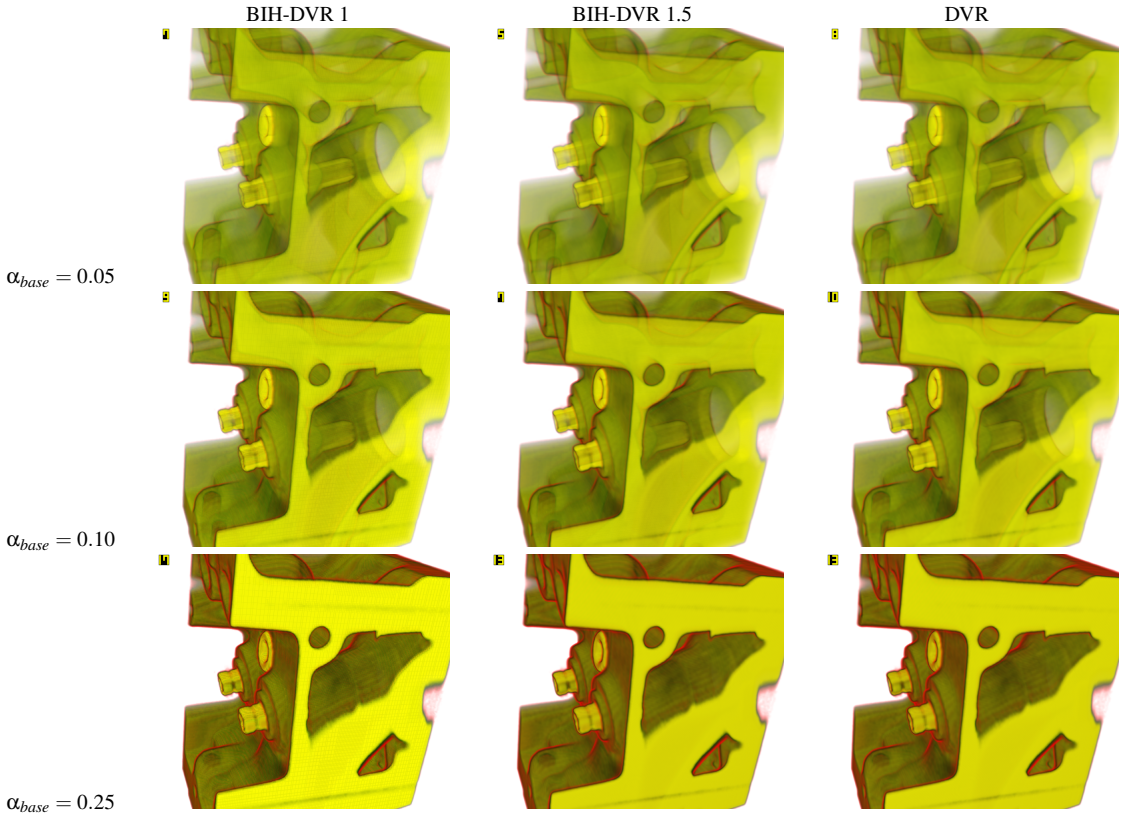


**Figure 5:** *Comparison of single-sample BIH-DVR to traditional DVR using the engine data set. In these screenshots, shading is disabled and material borders are not emphasized, hence the gradient is not used (except for approximating the density for BIH-DVR) and traditional DVR does not have to sample the local neighbourhood to compute the gradient.*
*We focus on the visual quality's dependency upon the overall opacity ($\alpha_{base}$ is the weight of a unit sample) and the grid's frequency. The heavy artifacts produced by BIH-DVR 1 are hidden quite well when using a low base opacity. The supersampled grid's frequency is significantly lower and hence BIH-DVR 1.5 can visually compete with traditional DVR. Performance is also comparable, although BIH-DVR profits more from early ray termination.*

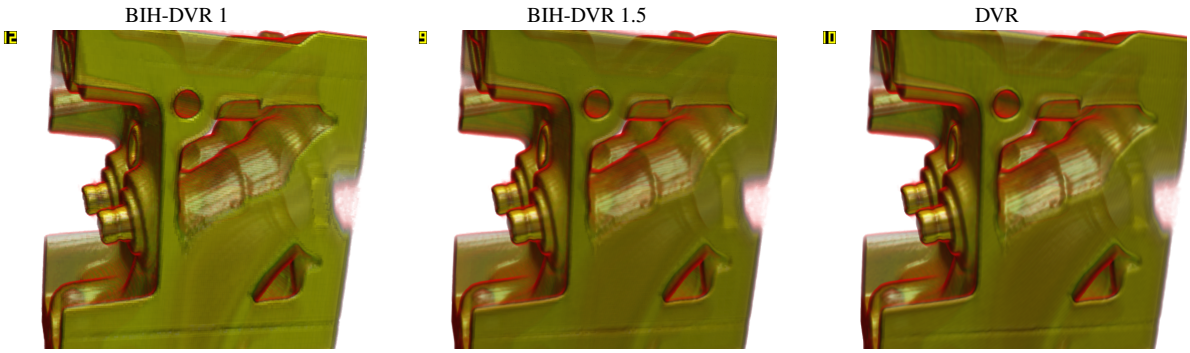


**Figure 6:** *Comparison of single-sample BIH-based DVR to traditional DVR using the engine data set, including shading and adjusting the opacity of a color contribution based on the gradient magnitude to emphasize material border surfaces. BIH-DVR 1 exhibits typical BIH artifacts due to imprecise density approximation. BIH-DVR 1.5 looks a lot smoother and is still rendered at 75% of the original BIH's speed although the LoD's tree is more than 3 times as large. The DVR reference exhibits some ring artifacts due to undersampling. The frame rates are comparable: 9-12 fps.*

The sources for the following engine renderings (figures 5 and 6) are summarized by table 4.

#### 6.2.2. The head aneurysm (vertebra) data set

The second data set is a rather large grid consisting in $512^3$ voxels. The volume is quite noisy; to remove most of the noise, we discarded all voxels whose density is below 12.5%. The effect of these additional 2.5% is illustrated in table 5. The volume's frequency is rather low, i.e. the higher resolution allows us to use the original grid for the BIH instead of a supersampled one. The renderings are presented in figure 7.

#### 6.2.3. The aneurysm data set

The next data set is another aneurysm. It is stored in a $256^3$ grid and is not very noisy. It resulted in very bad initial results due to high frequencies, therefore we tested two additional BIHs based on different supersampled grids. See table 6 and figure 8.

## 7. Conclusion

We analyzed the BIH data structure and found it to be a well-performing hierarchical partitioning scheme for sparse spatial elements. It combines many advantages of both the Bounding Volume Hierarchy and the *k*d-tree, resulting in a very memory-efficient structure with advantageous construction and traversal performance.

We have applied the BIH to uniform grids of voxel data, using voxels with a minimum density value as BIH elements and therefore discarding many noisy voxels. The sparse remaining voxels are partitioned hierarchically in a BIH structure. Neighboring similar voxels are merged when creating the BIH level-of-detail to be rendered, compressing the volume further. In most cases, the resulting BIH LoD's footprint is significantly smaller than the uniform grid's while providing implicit memory for additional attributes such as the precomputed gradient.

Rendering the voxels using Direct Volume Rendering involves traversal of the BIH on the GPU. We have presented an efficient and GPU-friendly memory layout for the serialized BIH tree. The traversal of a BIH is a recursive operation and hence needs a stack. Ordered traversal of a purely hierarchical structure for an arbitrary ray direction as required by volume ray-casting requires a stack anyway, so the primary advantage of the BVH (stackless fixed-order traversal) cannot be exploited for most volume visualizations. We have shown that stack-based traversal on the GPU can be quite fast, although there is still vast room to optimize the GPU for such tasks too. The dynamic branching granularity still needs to be optimized and the shared cache needs to be more flexible to place the stack there, freeing registers to increase the number of parallel threads and hence hide memory latencies more effectively. The latter point is addressed by future Direct3D 11 compatible hardware (supporting compute shader model 5.0): the shared cache is doubled to 32 KB, every thread may access the full memory (as atomic operation if needed) and so-called append/consume buffers may be used to build and access lists and stacks.

By partitioning the voxels hierarchically, we lose the implicit connectivity information of the uniform grid. Therefore trilinear filtering and on-the-fly gradient computation is not possible anymore - every BIH element needs to be rendered independently. To allow for direct volume renderings, the voxels need to define a scalar density value accompanied by a gradient. The gradient is used for shading (for the whole leaf) as well as for approximating trilinear filtering of

| | BIH used in table 3 | BIH used for rendering | DVR |
|---|---|---|---|
| Resampling factor | 1 | 1 | 1 |
| BIH elements | 16,481,795 (12.28%) | 2,346,996 (1.75%) | - |
| BIH construction | 40,435 ms | 5,586 ms | - |
| Max LoD depth | 27 | 27 | - |
| Data structure size | 93,218 KB | 22,152 KB | 262,144 KB |

**Table 5:** ***Vertebra data set****: comparison of the BIH based on the upper 90% of the density range and the one based on the upper 87.5% which has been used for rendering. The LoD requires less than a tenth of the uniform grid's size.*

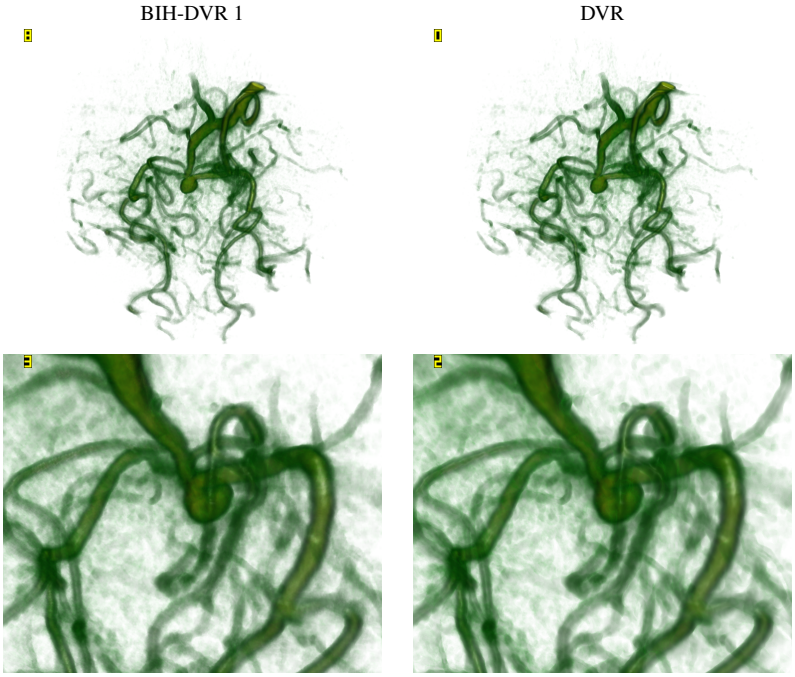


**Figure 7:** ***Vertebra data set****: comparison of single-sample BIH-based DVR to traditional DVR. The top image is a rendering of the full volume, the bottom image of a zoomed central region. Differences are hard to spot, and the BIH is clearly rendered faster, although it still only achieves interactive rates when zooming.*

| | BIH-DVR 1 | BIH-DVR 1.5 | BIH-DVR 2 | DVR |
|---|---|---|---|---|
| Resampling factor | 1 | 1.5 | 2 | 1 |
| BIH elements | 113,453 | 409,511 | 976,669 | - |
| BIH construction | 267 ms | 930 ms | 2,221 ms | - |
| Max LoD depth | 24 | 27 | 27 | - |
| Data structure size | 1,841 KB | 5,944 KB | 13,079 KB | 32,768 KB |

**Table 6:** ***Aneurysm data set****: comparison of different BIH sources for the renderings.*

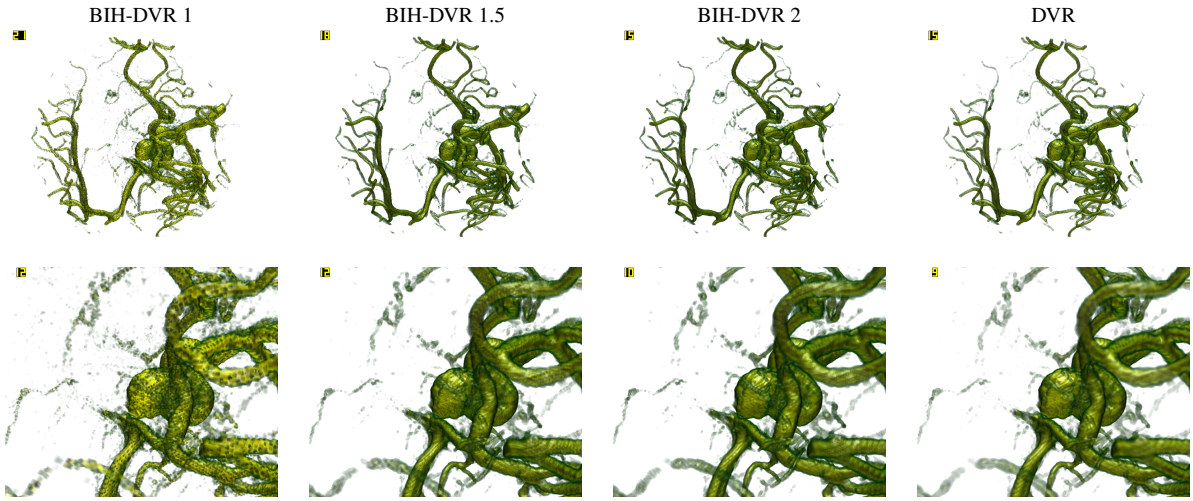


**Figure 8:** ***Aneurysm data set****: comparison of single-sample BIH-based DVR to traditional DVR using a high base opacity. The top image is a rendering of the full volume, the bottom image of a zoomed central region. The effect of basing the BIH on supersampled versions of the original grid is evident. The performance of BIH-DVR 2 is almost identical to traditional DVR and visual quality only slightly worse, but its LoD requires only 40% of the uniform grid's size.*

the density values inside the voxel. The precomputed 16 bit gradient can be stored for free in our BIH implementation. Our simple scheme does obviously not work well for high-frequency data. In that case, we have shown that by super-sampling the source grid (using trilinear filtering) and using the interpolated voxels as BIH elements, the visual quality can be greatly improved.

The performance of the presented BIH-DVR method surpasses traditional DVR in case the volume is very sparse, e.g. for aneurysm data sets (in some scenes by more than a factor of 8). More dense data sets are usually rendered somewhat slower on current GPUs, although the rates are still well-interactive for moderately sized data sets and view ports. The primary advantage though is the vast potential for memory footprint reduction, allowing to render very detailed, sparse grids on the GPU which would otherwise not fit into GPU memory or be rendered too slowly. The higher the grid resolution, the lower the frequencies, therefore the implicit degradation in visual quality compared to traditional DVR may be negligible.

The BIH-DVR method is just an example for possible BIH usages in volume applications. Instead of rendering blocks, one may use ellipsoids of different shapes and colors as BIH elements to render a volumetric particle system. A BIH may also be used to partition a volume adaptively in coarse blocks, especially if the volume is sparse so that empty regions can be eliminated implicitly.



## References

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), ACM, pp. 9–16.

[EWM08] EISEMANN M., WOIZISCHKE C., MAGNOR M.: Ray Tracing with the Single-Slab Hierarchy. In *Proc. Vision, Modeling, and Visualization (VMV'08)* (Konstanz, Germany, 10 2008), pp. 373–381.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d Tree GPU Raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[KAH07] KÄHLER R., ABEL T., HEGE H.-C.: Simultaneous GPU-Assisted Raycasting of Unstructured Point Sets and Volumetric Grid Data. In *Proceedings of IEEE/EG International Symposium on Volume Graphics 2007* (Prague, Czech Republic, 2007), A K Peters, pp. 49–56.

[TS05] THRANE N., SIMONSEN L. O.: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, Denmark, 2005.

[WK06] WÄCHTER C., KELLER E.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *In Rendering Techniques 2006 - Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.