Eurographics Symposium on Parallel Graphics and Visualization (2020)
S. Frey, J. Huang, F. Sadlo (Editors)

Finding Efficient Spatial Distributions for Massively Instanced 3-d Models

S. Zellmann[1], N. Morrical[2], I. Wald[3] and V. Pascucci[2]

[1]University of Cologne, Chair of Computer Science
[2]University of Utah, SCI, [3] NVIDIA

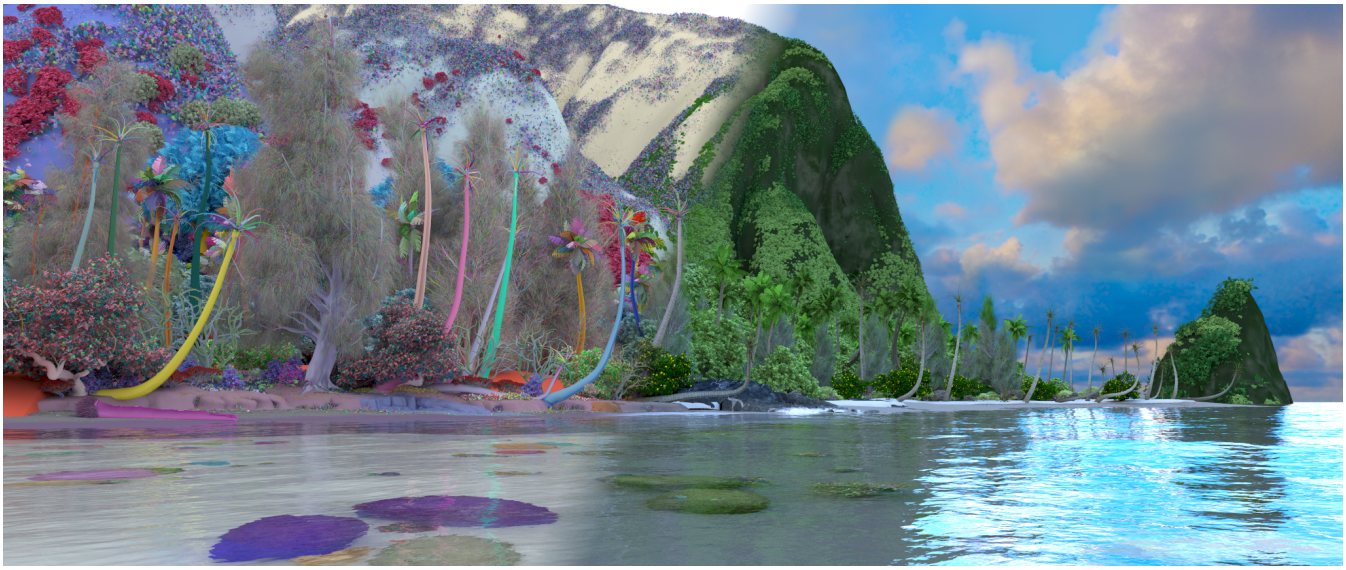


Figure 1: *Despite the heavy use of instancing in production assets like Disney's Moana Island Scene [TP18], the raw geometric complexity of these instanced meshes prevents many production assets from fitting within the memory of a single GPU. In the figure above, colors to the left encode the thousands of instance IDs used to decorate the island. For scenes like these, finding an efficient spatial distribution is nontrivial. However, our k-d tree construction algorithm can account for the various challenges that these types of 3-d models present in order to distribute this data efficiently.*

Abstract

Instancing is commonly used to reduce the memory footprint of massive 3-d models. Nevertheless, large production assets often do not fit into the memory allocated to a single rendering node or into the video memory of a single GPU. For memory intensive scenes like these, distributed rendering can be helpful. However, finding efficient data distributions for these instanced 3-d models is challenging, since a memory-efficient data distribution often results in an inefficient spatial distribution, and vice versa. Therefore, we propose a k-d tree construction algorithm that balances these two opposing goals and evaluate our scene distribution approach using publicly available instanced 3-d models like Disney's Moana Island Scene.

CCS Concepts

• ***Computing methodologies*** → ***Ray tracing;*** *Self-organization;*

1. Introduction

Instancing is a particularly useful technique when 3-d models contain *recurrent* structures that are geometrically complex and are replicated thousands of times. Typical examples are outdoor scenes that contain grass and foliage, like the Moana Island shown in Figure 1. By storing these complex geometric structures only once and then later reusing these structures several times by using an ID and an affine transformation to describe where to place an instance, studios can reduce the overall memory of an entire shot significantly. Despite the memory savings, some instanced 3-d models

DOI: 10.2312/pgv.20201070

are still so extensively complex that data-parallel rendering on a distributed memory system or a multi-GPU workstation should be taken into consideration. While in the field of scientific visualization, where data sets usually consist of a large amount of raw triangles, data-parallel rendering is commonly used, we are not aware of a large corpus of research papers focusing on data-parallel rendering of massively instanced production-quality models. While other options like resource loading on demand or data-parallel rendering of subsets were explored in the literature [CFS*18, BPR02], to our knowledge, our paper is the first one that extensively studies this problem from a scientific point of view.

Unfortunately, finding a data partitioning to distribute these instanced models across compute nodes is much more challenging than finding partitionings to distribute non-instanced models. A good data partitioning that is aware of instancing will have the following properties:

Even data distribution: It is generally desirable to find a relatively even data distribution (i.e. distributing roughly the same number of *entities* to each compute node) to make good use of the available memory and compute resources.

Even spatial distribution: Ray tracers bin rays to the spatial domain associated with the compute nodes. Irregular *spatial* distributions that potentially contain much empty space will result in load imbalances, ray replication, and bad rendering performance in general.

Little object overlap: Even with spatial decompositions that we use in this paper, the object bounds potentially reach beyond the spatial domain they are assigned to. Ray tracers would potentially intersect the object twice if it is present in multiple domains, so that little overlap and volume of object bounds is desirable.

Minimal replication: for the distributed rendering of *instanced* 3-d models in particular, should an instance be assigned to more than one node, the base mesh that instance refers to must be replicated. Ideally, base meshes should be replicated as little as possible.

Unfortunately, these various objectives and quality criteria potentially counteract each other. An even spatial distribution could, for example, be achieved when the whole mesh data is replicated on all nodes. A spatial partitioning scheme could then use a simple heuristic like *middle split* to equally distribute *space* among nodes; however, then the storage benefit of using instances would be obsolete. Similarly, an equal data distribution that could be obtained with a *median split* does not necessarily lead to a good spatial distribution.

Therefore, we propose an instance-aware k-d tree construction algorithm that balances the above opposing goals. Our data partitioning algorithm produces efficient spatial distributions for these massively instanced 3-d models while also being memory-efficient. In particular, the contributions we present in this paper are the following:

- A top-down k-d tree construction algorithm that is aware of instancing and that can, for example, be used in conjunction with a data-parallel ray tracer.
- A split heuristic that places an emphasis on balancing and reducing the overall memory consumption and that exposes a number of parameters that can be fine-tuned.

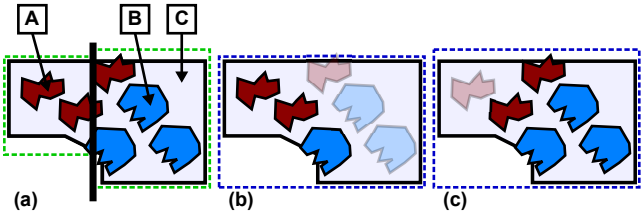


Figure 2: *Spatial distribution and overlapping object bounds. (a) shows a 3-d model consisting of three instanced object types A (3x), B (3x), and C (1x) and a potential spatial split. The dotted green lines indicate the domain bounds of the respective subtrees. (b) shows the instances assigned to the left subtree, with the object bounds of the contained instances indicated by a blue dotted line. (c) shows the instances assigned to the right subtree. Note that while the domain bounds do not overlap, the object bounds have almost the same size as the 3-d model is dominated by a large base mesh. Also note how the second from left A-type and the first from left B-type instances straddle the splitting plane and are replicated in both subtrees.*

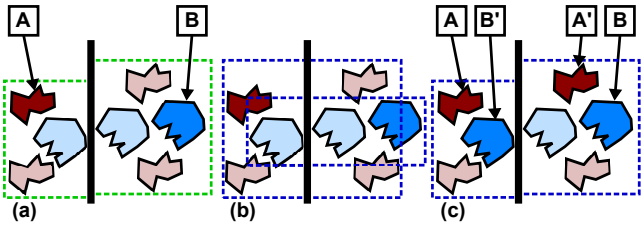


Figure 3: *Spatial vs. object partitioning. (a) shows the domain bounds for the spatial split indicated by the green vertical line. (b) shows a naïve object space distribution like that found by a typical bounding volume hierarchy builder that is not aware of instances but only of base meshes. (c) shows how the object bounds according to the split have significantly less overlap because the base meshes of type A and B are replicated. Our method aims at finding good spatial distributions with minimally overlapping object bounds and is also designed to minimize memory overhead due to replication.*

- A novel, heuristical approach to distribute large base meshes that does not actually split those structures until an advantageous split plane was found.

We evaluate the constructed data structure with several massively instanced 3-d models like Disney's Moana Island Scene [TP18].

2. Background

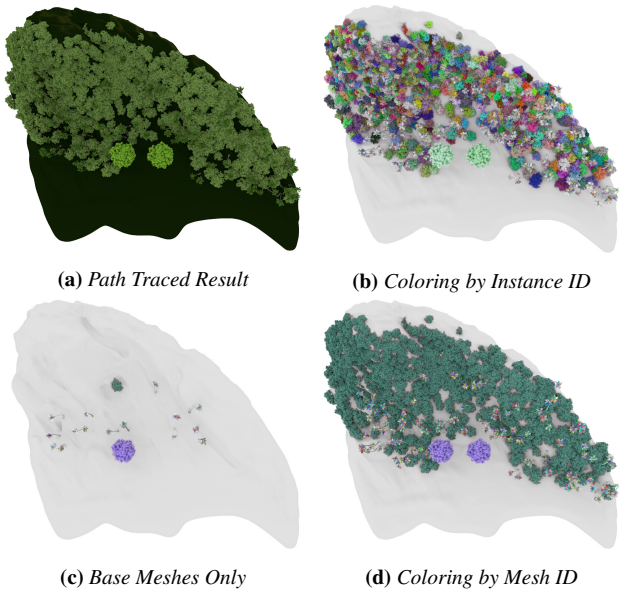
The approach we take in this paper is top-down k-d tree construction, where we recursively split the space occupied by the 3-d model. When performing a spatial split of the domain occupied by an inner node, it is crucial to find a beneficial split position. We choose a greedy approach with heuristics that find a local optimum regarding the node that is currently being split.

We adopt the following terminology. We refer to the geometry that is instanced as *base meshes*. We call the axis-aligned bounding boxes that surround individual instances (or surround the union

of several instances) *object bounds*. The non-overlapping regions associated with each leaf node we call the *domain bounds*. Note that the domain bounds of the leaf nodes are a full space decomposition of the 3-d model. In Figure 2, the object bounds and the domain bounds are represented with blue and green dotted lines, respectively.

Finding efficient data partitionings for instanced models poses a number of challenges. Large outdoor scenes, for example, often contain terrain mesh instances that span the whole 3-d model but are instanced only once. In Figure 2, the object bounds of the large base mesh associated with instance type *C*, for example, are much larger than the domain bounds of the two potential nodes, which leads to this type of mesh dominating the decision where to perform the split. Often, the smaller instances are just fully contained inside the object bounds of those large base meshes, and with a naïve approach would not contribute to the splitting decision at all.

Another challenge with instanced models is the fact that smaller instances are often spread out all over the model (cf. Figure 3). A typical bounding volume hierarchy (BVH) builder that is only aware of base meshes and not of instances would find an *object partitioning* like the one in Figure 3 (b) which would group the base meshes of type *A* and *B* and their respective instances into separate tree nodes. By replicating the base meshes (giving us *A'* and *B'* in Figure 3 (c)), the object bounds are much tighter (in this example they are actually identical to the domain bounds). The method we propose in this paper aims at finding spatial distributions like the one in Figure 3 (c) while still trying to keep memory overhead and data replication as low as possible. An example of the various challenges illustrated in Figure 2 and Figure 3 can be found in Figure 4.



(a) *Path Traced Result* **(b)** *Coloring by Instance ID*

(c) *Base Meshes Only* **(d)** *Coloring by Mesh ID*

Figure 4: Mountain A *structure of Moana Island. This part of the 3-d model is a good example of the difficulties encountered with data-parallel instancing. While the terrain mesh dominates the whole spatial extent, the bushes are spread out over the model and will likely be replicated on many nodes if not explicitly accounted for. The two large trees in the center of the lower half of the images are especially challenging and many spatial split heuristics would just replicate them on multiple nodes.*

3. Related Work

Geometry instancing was traditionally used in 3-d hardware to reduce both the storage overhead as well as the bandwidth required to transmit geometry to the GPU, and to minimize the number of state and texture changes due to rendering the same triangle over and over again [Car05]. Nowadays, many industry-strength ray tracing libraries and production rendering systems make use of instancing [WWB*14, PJH16, GIF*18], where the primary motivation is to reduce memory overhead while also allowing for more efficient artistic workflows. Typical applications are large outdoor environments containing tree leaves and foliage [DCSD02, BLZD12, WLT13]. One very early application of real-time ray tracing of a (then) massively instanced model is that of Wald [Wal04], who reported interactive frame rates of about 7 fps at VGA resolution on a cluster with 24 dual-core CPUs. Johansson [Joh09] used hardware instancing to improve the rendering performance of large building information models. Santos and Filho [SF14] showed the effectiveness of instancing when applied to large CAD models. Using shape matching, they were able to reduce the storage consumption of a CAD model to only 5 % of the original 3-d data set. Slomp et al. [SDF*13] used instancing to splat large point clouds using a variety of different shapes.

Out-of-core rendering approaches often use space partitioning to distribute objects across nodes when the data set as a whole does not fit into the memory of the GPU or a single rendering node. Kontkanen et al. [KTO11], for instance, used an octree to organize point data for global illumination. Interactive out-of-core ray tracing of large 3-d models with a binary space partitioning tree was explored by Wald in [WDS04]. In this work, Wald used memory-mapped file I/O and relied on the operating system's disk cache to keep a relevant working set of the 3-d model in main memory.

Visualization frameworks often employ a sort-last parallel rendering approach [MCEF94] and fix the data distribution across frames. Especially with homogeneous workloads like direct volume rendering, both binary space partitioning and *k*-d trees are a popular choice for distributing the data, where renderers then sort the intermediate rendered images in visibility order before compositing the partial images together [MSE06, MMD06, MM10]. Abram et al. [ANG*18] recently presented an asynchronous, massively parallel ray tracing system that runs on supercomputers. Their system assumes that the data is already distributed on a uniform grid and uses this knowledge to distribute rays among compute nodes. Work in this field was also conducted by Navrátil et al. [NFLC12, NCFL14] who investigated ray scheduling methods for distributed ray tracing. Son and Yoon [SY17] presented an out-of-core ray tracing software for multi-GPU systems which distributes data to compute nodes based on the resource demand of the scheduled task, and amongst others based on a time estimate for the data transfer to the compute node.

Finding efficient spatial or object splits to subdivide a group of primitives is crucial for ray tracing efficiency. Both *k*-d trees

```
// Build k-d tree with fixed 2**N leaves from
// from instances and precomputed base mesh SVT
void makeTree(Instances inst[], SVT svt) {
    int numLeaves = NUM_COMPUTE_RANKS;

    // The root domain bounds are just the
    // object bounds of all the instances
    aabb domain(inst, svt);
    Node root = makeInner(domain, inst);

    doSplit(root, inst, svt, numLeaves);
}

// Top-down recursive splits
void doSplit(Node n, Instances inst[],
             SVT svt, int depth) {
    if (depth == 1)
        makeLeaf(n.domain, inst);
    else {
        // Find a split for the domain bounds
        Split s = split(n.domain, inst, svt);

        // As an optimization, try to shrink the
        // child domain bounds by intersecting them
        // with their object bounds
        aabb objL(s.instL);
        aabb objR(s.instR);
        s.domainL = intersection(s.domainL, objL);
        s.domainR = intersection(s.domainR, objR);

        Node n1 = makeInner(s.domainL, s.instL);
        Node n2 = makeInner(s.domainR, s.instR);

        doSplit(n1, s.instL, svt, depth >> 1);
        doSplit(n2, s.instR, svt, depth >> 1);
    }
}
```

Figure 5: *Top down k-d tree construction. The k-d tree's leaf node count is fixed to the number of (2^n) compute ranks.*

[WH06] and bounding volume hierarchies [AL09] can be used to reduce the complexity of an intersection query to $O(\log n)$ over the number of primitives. Bounding volume hierarchies especially are very popular because their size can be estimated a priori, and because they do not necessarily need to be fully rebuilt when the scene changes. Many ray tracing frameworks build both data structures using the surface area heuristic (SAH) [GS87]. A whole research branch has recently focused on finding a good balance between tree quality and tree construction performance. On GPUs, bottom-up construction schemes are popular. Very fast builders use Morton codes to construct trees with a naïve middle split [ME17, LGS*09]. More recent work has focused on clustering of nodes, and construct a BVH from the bottom up while minimizing surface area in a local neighborhood [GHFB13, MB18].

To the best of our knowledge, no scientific work has so far *directly* concentrated on the problem of finding efficient data distributions for massively instanced 3-d models.

4. Spatial Index Construction

A reasonable choice to distribute a 3-d model among several compute nodes is to use k-d trees [PJH16]. In contrast to building k-d trees to accelerate ray tracing, in our case, the number of leaf nodes that we will distribute the scene data on is known a priori to be the number of compute nodes. With a k-d tree, we can also sort the data in visibility order, which is beneficial for some rendering algorithms. Therefore, we store instances at the leaf nodes, which consist of an id as a reference to their respective base mesh, an affine transformation matrix that describes how to orient, position or scale the instances, and some auxiliary data such as pre-transformed bounding boxes.

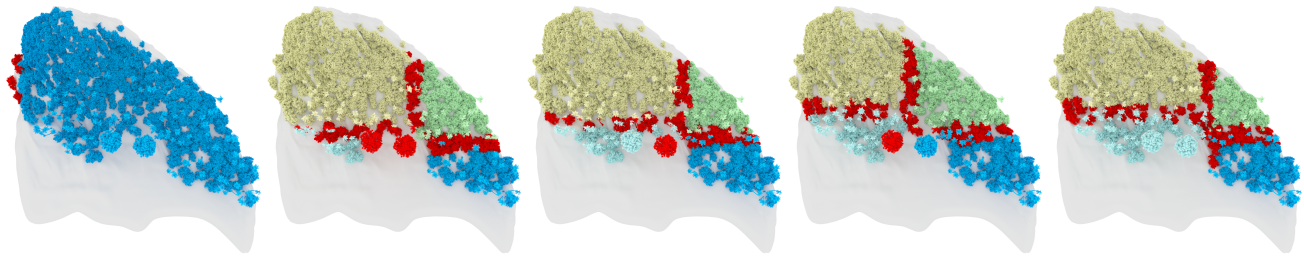
4.1. Finding optimal splits

We start out by splitting the domain bounds of the root nodes, which also happen to be the root node's domain bounds. We then recursively split the domain bounds until the number of desired leaf nodes is reached. We use greedy top-down construction to build a fully balanced binary tree with 2^n leaf nodes, which is also the number of compute node ranks we want to distribute the scene to. This restriction is specific to our application but could be easily relaxed by building balanced, but not *fully* balanced trees. We use binning with 64 bins per potential split to accelerate the tree construction algorithm. The primitive type we consider during split computation is the instance, i.e., except for large single-instance base meshes that we handle separately, we never split instances that straddle candidate planes into individual triangles.

As an optimization, before we evaluate split candidates, we try to shrink the domain bounds by computing their intersection with the object bounds of the instances contained inside. When testing split candidates, we use a cost function to test for local optimality that we describe in the following. This whole procedure is summarized in Figure 5.

In the absence of instancing, we can find a split that distributes space equally among compute nodes by using the *middle split* heuristic, which splits the bounding box in half along a given split axis. Likewise, we can find a split that evenly distributes data by using the *median split* heuristic, which distributes the same number of primitives to the left and right child nodes.

However, in the presence of instanced geometry, implementing these heuristics requires some additional consideration. When accounting for instanced geometry, we could implement these two heuristics using one of the following strategies. First, we could have our chosen split heuristic only consider *base meshes*, and simply ignore instancing. We would then need to distribute the instances derived from these base meshes to the corresponding sibling nodes that these base meshes were assigned too. This strategy would give us the configuration depicted in Figure 3 (b), where the object bounds and the underlying base mesh bounds guiding the heuristic differ significantly. With that strategy, a split that would distribute the instances in a favorable way would be hard to find. Alternatively, we could have our chosen split heuristic only consider *instances*. In this work, we conversely decided to only consider the instances when testing split candidates. This will give us a relatively easy way to find spatial splits, but at the cost of potentially having to replicate instances and base meshes across nodes. As neither the middle split nor the median split heuristic are aware of instances and replication, we propose to combine them with two new heuristics:



(a) *min-rep split* **(b)** *equal-geo split* **(c)** *median split* **(d)** *middle split* **(e)** *combined* (5 : 1 : 1 : 1)

Figure 6: *Examples of the splits that the respective heuristics would generate in isolation. The colors show a data assignment to four nodes. We assigned the color gray to the large base mesh for better visual reference. Instances that were assigned to more than one node are colored in red. Note how only our combined heuristic assigns the two large trees in the middle to a single node while retaining a well-distributed layout.*

- The *min-rep split* heuristic finds a split by minimizing the number of triangles, from underlying base meshes, that we might replicate across the nodes. This *min-rep split* can be achieved by considering a split through the instances, distributing the base meshes accordingly, and summing up the number of triangles from base meshes on each side of the split plane.
- The *equal-geo split* heuristic is relatively analogous to a median split, only that the split is driven by the instances, while *the costs of the split* are computed based on their base meshes.

On its own, these heuristics may run into several corner cases. The *min-rep split* heuristic, when used in isolation, would often distribute all instances to one node, as this would minimize base mesh replication. The median split heuristic, when considering only the instances as actual data, could produce a relatively imbalanced data distribution, since it does not consider the base mesh data distribution as the *equal-geo* split heuristic does. Figure 6 illustrates some of the problems when the heuristics are applied in isolation.

Therefore, we propose to combine these four heuristics using a linear cost function that applies weights to the *normalized* values calculated for each heuristic:

$$C = w_1 \frac{T}{T_{max}} + w_2 \frac{|T_l - T_r|}{T_{max}} + w_3 \frac{|mdn - s|}{b} + w_4 \frac{|mdl - s|}{b}, \quad (1)$$

where w_1 is the *cost related* weight associated with the min-rep split heuristic, which is normalized by dividing by the maximum number of triangles from instanced base meshes T_{max}. Conservatively setting T_{max} to twice the number of triangles of the base meshes accounts for the unlikely worst case that all base meshes are replicated on both sides.

The weight w_2 is associated with the equal-geo split heuristic and balances, though driven by the instances, the number of base mesh triangles T_l and T_r on either side of the split plane. As this is also a geometry-related heuristic, we again divide by the same T_{max} we also used for min-rep split.

w_3 and w_4 are associated with the median split and the middle split heuristic, respectively. The two heuristics are however also normalized so they are roughly within the same range as w_1 and w_2. We therefore divide the results from the heuristics by the extent b of the object bounds along the split axis.

Greedy algorithms that find an optimal spatial index for ray tracing often use the Surface Area Heuristic (SAH). Because our spatial distribution is not necessarily limited to ray tracing, we chose not to incorporate this specific heuristic. If desired, we could, however, include SAH in our combined heuristic. Regardless of the heuristic used for the top-level splits, a typical ray tracing pipeline will likely build a SAH BVH *at the node level* to accelerate local ray traversal.

4.2. Handling large geometric structures

Although so far we have devised a combined splitting heuristic capable of minimizing base mesh replication, we still must replicate any split-plane-straddling base meshes to the left and right nodes. This base mesh replication can be problematic for scenes that contain large, single instance base meshes like the terrain mesh in Figure 1 or the type C instance in Figure 2. In many scenes, base meshes like these can consist of hundreds of thousands of individual triangles, and should still be split and distributed. Thus, we make an exception to our previous rule, where we chose not to split individual instances, to now allow splits for large single instance base meshes.

Initially, we sort all instances by the reference count of each instance's corresponding base mesh. As a rough heuristic, we can identify these splittable instances as those whose base mesh's corresponding reference count is one. Alternatively, we could choose a more sophisticated test, like checking if the base meshes' extents are sufficiently large, for example. For our test data sets, though, we found this simple reference count heuristic to work reasonably well. In our large landscape scenes, base meshes that are referenced only once are usually large, and conversely, base meshes that are referenced hundreds and thousands of times are usually tiny.

We assume that eventually, i.e. *after* the k-d tree was built, we are going to split the large base meshes at the domain bounds and distribute them as single instance meshes to their respective leaves. Our instance-aware construction algorithm should therefore take those *potential* instances into account and therefore needs to determine their object bounds and respective triangle counts to compute and normalize the heuristic. Knowing the bounds and number of triangles, when naïvely implemented, would require us to split the

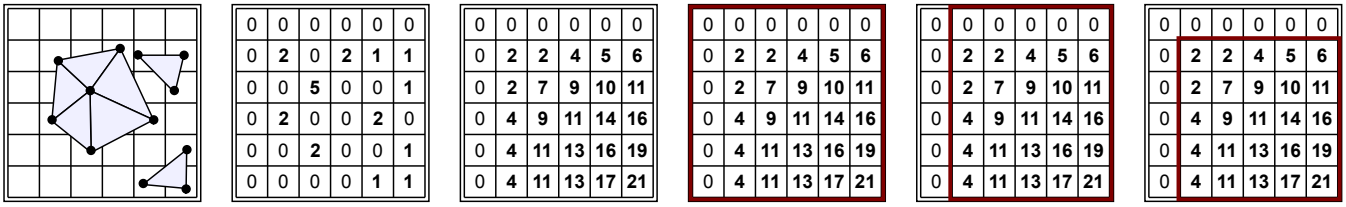


Figure 7: *Building an SVT (here we demonstrate the concept using a 2-d summed* area *table) over triangle vertices and using it to compute the object bounds from given domain bounds. Vertices that are shared by n triangles count as n rather than a single vertex. With the SVT, one can find the object bounds within the domain bounds by successively moving the sides of the bounds inwards. When the vertex count inside the a potential object bounding box is lower than the vertex count inside the domain bounds, we know that we have found an invalid configuration.*

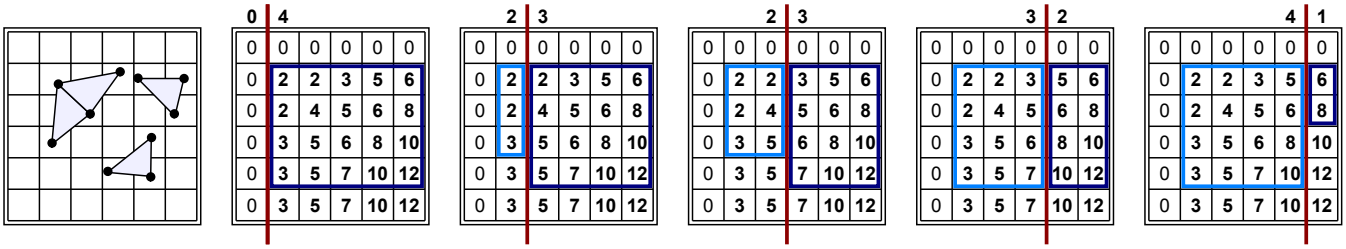


Figure 8: *Calculating* rough *object bounds and triangle count estimates using SVTs for five split plane candidates. The SVTs do not store the connectivity information between the triangle vertices; it is thus not possible to* exactly *determine which triangles partially overlap a spatial domain. The algorithm rather computes the number of triangle vertices contained inside the domain, as well the object bounds around those vertices; the triangle count estimate (put to the left and right of the split plane) is roughly estimated from the vertex count. An approximation is sufficient because it is only used to drive a k-d tree split heuristic and not to distribute actual data.*

base meshes on a triangle level over and over again, which is prohibitive performance-wise.

We therefore propose to split the large single instance base meshes only virtually, and only when the *k*-d tree is fully built up we actually split the meshes into individual instances. For the virtual splits, we use a summed volume table (SVT) to compute the instances' object bounds and triangle counts. SVTs are uniform, rectilinear grids that store at each grid cell the 3-d prefix sum with respect to the origin of the grid.

We first project all the triangle vertices of the base meshes to a source grid that we later build the SVT from. Each grid cell will store the number of vertices contained within. Vertices that are shared by *n* triangles are treated as *n* vertices instead of one. From that initial grid, we build the 3-d prefix sum over the vertex counts, which gives us the SVT.

Given a split candidate, we can now feed the domain bounds to the left and the right of that into the SVT, which will allow us to query the number of all the large single instance base mesh vertices within the domain, as well as their object bounds. As the connectivity between triangles is lost, this is only a rough estimate for the object bounds of the actual triangles.

The advantage of this approach is that object bounds and vertex count inside the domains can be found in logarithmic time [VMD08, ZSL18]. The procedure involves starting out with the domain bounds, counting the number of vertices contained inside, and then shrinking the domain bounds towards the object bounds. If shrinking was successful or not is determined by tracking the vertex count. If the vertex count within the shrunk bounds is below that of the domain, shrinking was an invalid operation. Querying the vertex count is a constant time operation with SVTs. The triangle count that we require to normalize the geometry-related heuristics we approximate by simply dividing the vertex count by three. While being somewhat inexact, we find this approximation acceptable because we only need it to later feed the result into a heuristic, and do not use the count to actually distribute any data. This overall procedure is illustrated in Figure 7 and Figure 8.

A common problem with SVTs or their 2-d equivalent—summed area tables—is that the prefix sum operation can overflow. It is thus in general necessary to have the SVT use an integer type with more bits than that of the source grid. A property of the SVTs that we compute is however that they are sparse; i.e. the total sum stored in the SVT will just be the number of all triangle vertices from large base meshes, which is known a priori and also determines how many bits we need for the SVT elements. Assuming that we never have more than roughly four billion vertices, this allows us to use 32-bit unsigned integers for the SVT elements. Sensible choices for the SVT size depend on the data set size. For all our test cases we us an SVT of size 256^3. Binning will discretize the domain of the root node level anyway, so that a much higher resolution would not be necessary.

Only when we have fully built up the *k*-d tree, we split the large base meshes into their domain bounds and distribute the resulting geometry and instances to the respective leaves. We therefore first

```
Split split(Node n, Instances inst[], SVT svt) {
    aabb domain = n.domain;
    int axis = argmax(domain.w,domain.h,domain.d);
    aabb obj(inst);  // object bounds for median
    sort(inst,axis); // also for median split
    Split bestSplit = {};
    Instances instL[], instR[]; // empty sets

    foreach (bin in bins) {
        float S = domain.min[axis] + bin * BinWidth;
        // Distribute large base meshes to either
        // side of split candidate S using SVT
        addVirtualInstances(inst, svt, S);

        foreach (I in inst) {
            if (bounds(I).max[axis] < S)
                instL.append(I);
            else if(bounds(I).min[axis] >= s)
                instR.append(I);
            else {
                instL.append(I);
                instR.append(I);
            }
        }

        float mdl = (domain.min[axis]
                    +domain.max[axis])/2;
        float mdn = bounds(inst[inst.size()/2]);
        float Tl = instL.countTriangles();
        float Tr = instR.countTriangles();
        float T = Tl+Tr;
        float b = obj.size[axis];
        float Tmax = inst.countTriangles()*2;

        // Determine the costs associated with split S
        float C = w1*(T/Tmax) + w2*(abs(Tl-Tr)/Tmax)
                + w3*(abs(mdn-S)/b) + w4*(mdl-S)/b;

        if (C < bestSplit.cost)
            bestSplit = { C, S, instL, instR };
    }

    aabb domains[2] = splitInHalf(domain);
    bestSplit.domainL = domains[0];
    bestSplit.domainR = domains[1];

    return bestSplit;
}
```

Figure 9: *Split function for top-down k-d tree construction. We use a greedy heuristic with binning (64 bins) and compute the best split position as the weighted average of four individual split heuristics.*

partition the triangles of each base mesh to both sides of the respective split planes, duplicate those triangles that straddle the split plane, and thus obtain up to one new mesh per leaf node. This mesh we assign an instance that is initialized with the transform of the original single instance.

The pseudocode in Figure 9 summarizes the split routine including how we compute the split heuristic and when we use the SVT to determine virtual instances.

5. Results

We perform a thorough analysis of our *k*-d tree construction algorithm using several massively instanced, publicly available 3-d models. We are interested in the storage overhead of our algorithm as compared to the middle split heuristic and the median split heuristic. We, therefore, construct *k*-d trees with all three heuristics and calculate the total storage costs as the sum of the size of all triangles and all instances.

The size in bytes of a triangle or an instance will vary from application to application. We integrate our prototypical implementation into the Visionaray framework [ZWL17] and load models in the pbrt format that we integrate using the `pbrtParser` library [Wal]. We make the application-dependent assumption that triangles have a memory requirement of 32 bytes per vertex (12 bytes position, 12 bytes vertex normal, 8 bytes texture coordinate) for a total of 96 bytes per single triangle. We further assume that we can store an instance using 128 bytes total (64 bytes for the transformation matrix, 24 bytes for the pre-transformed bounding box, and 40 extra bytes for bookkeeping information like instance id, or padding). Other applications will assess the memory requirements differently; our estimate, for example, excludes space required for texture images, which would make a fair comparison much harder. Some applications might also store the inverse transformation matrix along with each instance so that rays can be efficiently transformed into object space. For our tests, we use the following data sets that are also depicted in Figure 10.

PBRT Ecosys 141 base meshes, 1.17 M triangles total, 12.8 K instances, 114 MB memory with our cost estimate.

PBRT Landscape 3870 base meshes, 27.7 M triangles total, 408 K instances, 2.72 GB memory with our cost estimate.

Moana Mountain 1.09 K base meshes, 7.65 M triangles total, 2.30 M instances, 1.03 GB memory with our cost estimate.

Moana Island 1.46 M base meshes, 141 M triangles total, 66.5 M instances, 22 GB memory with our cost estimate.

We compare our method to the median split and the middle split heuristics. However, even with those, we employ the strategy to split large meshes into instances with an SVT, as the results without that optimization would not be as meaningful. We assign weights for the cost function that favor lower total storage costs, as this is one of the main goals of our method. We therefore assign the value 1 to each of the weights but the one that is associated with the min-rep split heuristic that we assign the value 5.

In a first test setup, we are interested in a qualitative assessment of our method. We therefore calculate partitionings for $N = 2, 4, 8, 16, 32, 64, 128, 256$ compute nodes. We compute the total memory requirement for the partitionings, as well as the sum of the object bounds of the N nodes and present an average taken for all the test models we test with, as well as confidence intervals based on the standard deviation in Figure 11. There we also report the average *range* of the metrics as mentioned earlier averaged over the four models, i.e. we report the difference between the maximum and minimum memory requirement and domain bounds volume on each node.

We perform a sensitivity analysis to find out how susceptible our heuristic is to parameter changes. Therefore, we single out each one of the weights and fix all the other weights to 1. We then vary the weight we singled out and compute partitionings for $N = 256$ nodes. We report the influence of the changes in weight on the metrics from before in Figure 12.

We also measure the speed of the *k*-d tree construction algorithm

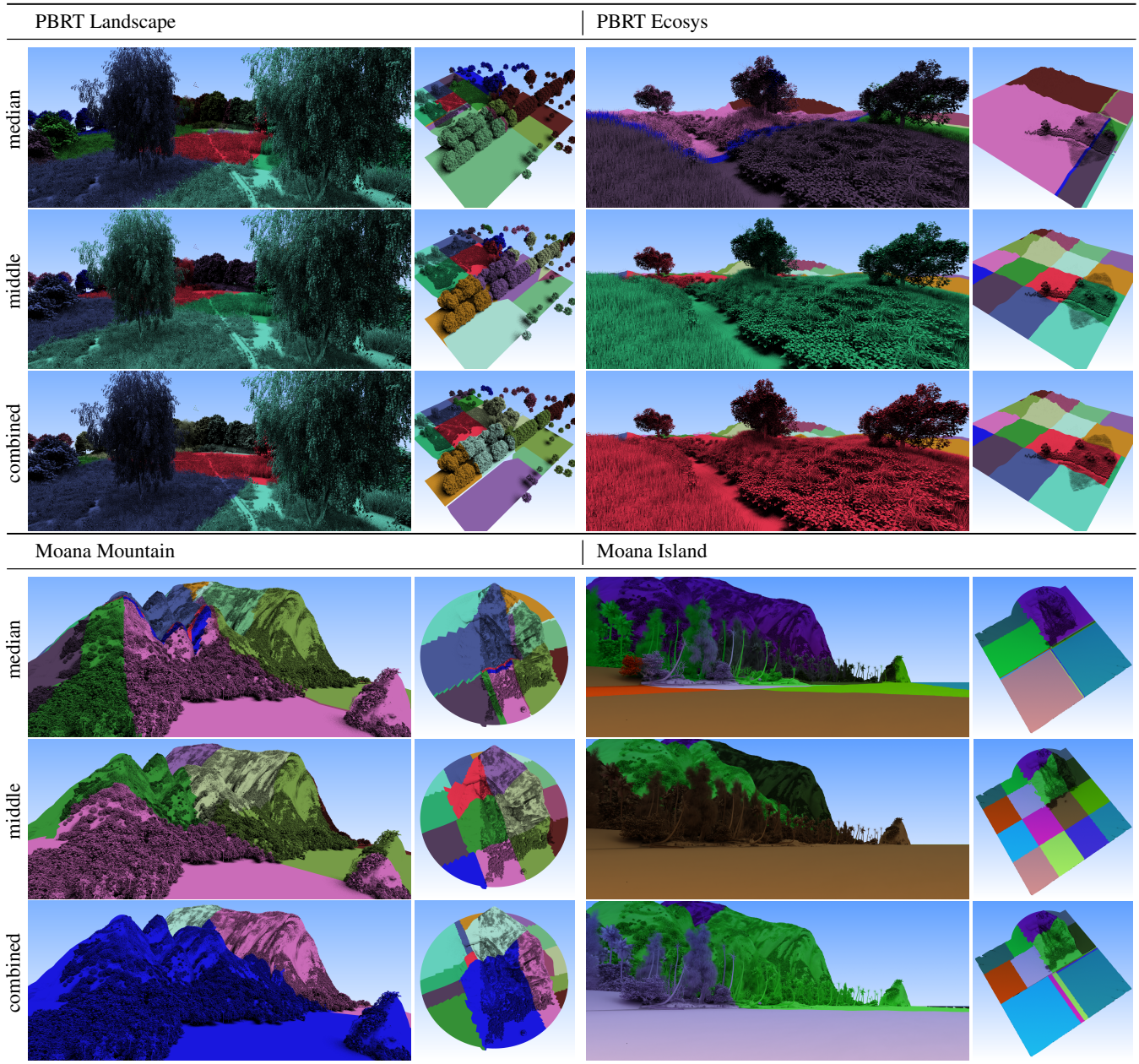


Figure 10: *Our test data sets, color coded by assignment to 16 compute nodes. We show spatial partitionings from the middle split heuristic, the median split heuristic, and the combined heuristic with the weights set to* $w_1 = 5, w_2 = 1, w_3 = 1, w_4 = 1$.

on an eight core Intel Xeon system with 128 GB DDR RAM. Of the eight cores we however only use one core so far because our implementation is single-threaded. We report results for partitionings of $N = 4, 32$, and 256 nodes. Our timing results are comprised of the SVT construction phase, the recursive, top-down tree construction phase, and the phase where we split and distribute large base meshes across the nodes. We report those results in Table 1.

As an indicator how the choice of heuristic would affect rendering performance, we feed the obtained paritionings into a prototypical multi-GPU renderer that performs path tracing using environment lighting. As it is hard to say how rendering performance is affected by the choice of top level split with such such a setup, we rather generate heatmap results that indicate how many ray transfers along one path will occur. We present those images for the PBRT Ecosys and Moana Mountain data sets for an assignment to $n = 16$ nodes in Figure 13.

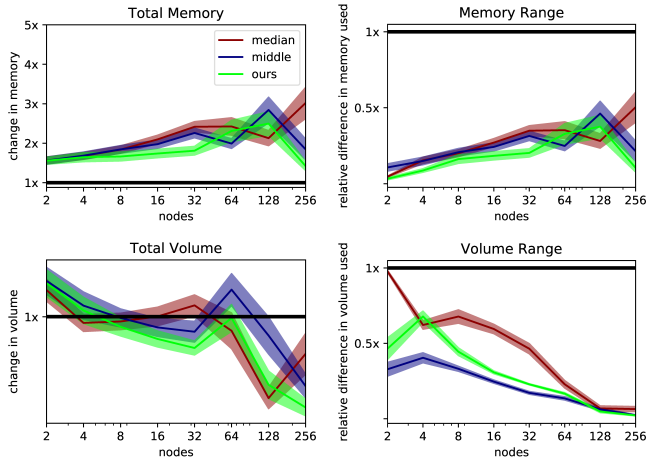


Figure 11: *Total memory and object bounds volume summed over all nodes, as well as memory range and volume range (difference of maximum and minimum per node). The values are averaged over the test models. Error bands signify the similarity of those measures across test models.*

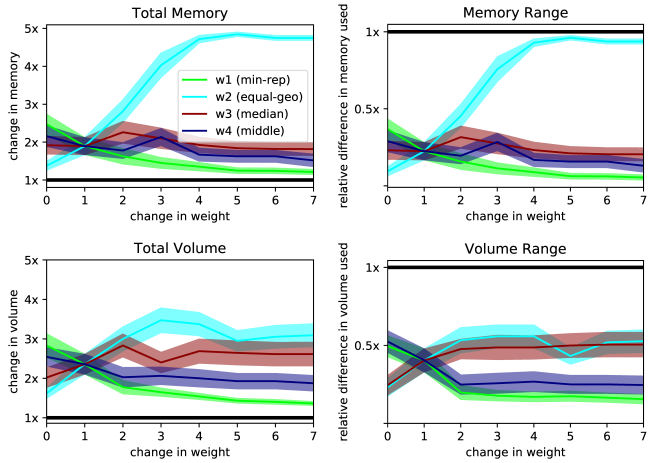


Figure 12: *Sensitivity analysis results. We fix all weights but a single one to* 1 *and vary that singled out weight in an empirically determined range. We report the weights' influence on total memory and on the total volume of the object bounds over all nodes, as well as on the memory range and the volume range (difference of maximum and minimum per node). The values are averaged over the test models. Error bands signify the similarity of those measures across test models.*

5.1. Discussion

In our results we have mostly focused on the min-rep heuristic that tries to reduce the total costs from the paritioning across all nodes, and the graphs and the images we created indicate that the heuristic is successful at that. With instancing even more than with simple triangle meshes there exist a number of quality criteria that contradict each other. Our sensitivity analysis indicates that the four

	Ecosys			Landscape			Mountain			Moana		
	Nodes			Nodes			Nodes			Nodes		
	4	32	256	4	32	256	4	32	256	4	32	256
SVT	0.156			0.782			0.153			7.565		
TD	0.025	0.067	0.137	1.528	3.126	4.310	4.198	12.07	21.48	595.6	1449.	2286.
Split	0.086	0.130	0.526	5.640	11.31	42.27	0.238	0.308	0.715	71.20	114.5	374.0
$\sum$	0.267	0.353	0.819	7.950	15.22	47.36	4.589	12.53	22.35	674.4	1571.	2668.

Table 1: *Timing results for the k-d tree construction algorithm in seconds, itemized by SVT setup, top-down construction, and large base mesh splitting.*

weights that can be used to fine-tune the construction algorithm are effective at what they are meant to do.

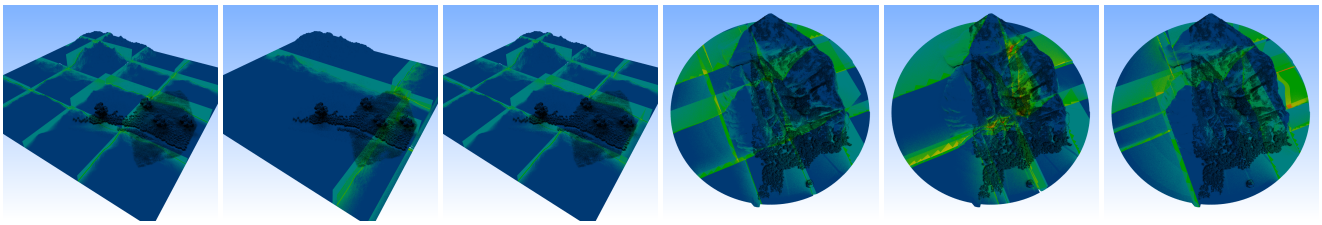
One objective that we explicitly not addressed was that of finding partitionings that lend themselves to good rendering performance. We assume that the renderer will build an optimal rendering data structure, e.g. a BVH with Surface Area Heuristic, on the node level of our trees. We still find it reassuring that our heuristic will find partitionings for which our multi-GPU renderer will not excessively transfer rays between nodes. Our heuristic in most cases even performs a bit better in this regard than the other two heuristics.

The decision to prefer spatial partitioning over object partitioning and thus *k*-d trees over BVHs was one that we made explicitly. It was driven by the thought that rays are more easily binned according to the domain bounds of the *k*-d tree leaf nodes than to the potentially overlapping object bounds of a BVH. It would still generally be possible to use object partitioning to address the various challenges with instanced 3-d models, and it would be interesting to compare such an implementation with our approach. Effectively, with object partitioning, one would not have to deal with the geometry replication problem but instead with replicating rays across nodes. A problem with spatial subdivision is that for large numbers of ranks, the greedy heuristic sometimes just will not find a split, resulting in very few nodes getting no work assigned at all. This problem could, for example, be solved by using object splits.

6. Conclusions and Future Work

We have presented a heuristical *k*-d tree construction algorithm for massively instanced 3-d models. The split heuristic we employ is designed to find a good trade-off between even spatial distributions on the one hand, and an even data distribution on the other hand. In contrast to data structures aimed at 3-d models without instances, the construction algorithm has to treat both the geometry as well as the instances themselves as input. The user can tune the construction algorithm by adapting weights and can decide for either of the objectives to be more important than the others. 3-d models with instances often contain larger structures like terrain meshes that are instanced only once, which is specially handled by our construction algorithm. Terrain models, especially if they are procedurally generated, often contain millions of triangles. We decided to keep the construction costs moderate by extracting virtual instances from the terrain model while splitting.

Part of the reasoning not to handle individual triangles in the construction pipeline is that we can in the future devise data-parallel



(a) *middle split* **(b)** *median split* **(c)** $w_{1,2,3,4} = \{5,1,1,1\}$ **(d)** *middle split* **(e)** *median split* **(f)** $w_{1,2,3,4} = \{5,1,1,1\}$

Figure 13: *Heatmap visualization of the number of ray transitions a distributed path tracer would require to perform for the middle heuristic, the median heuristic, and our combined heuristic with weights $w_1 = 5$, $w_2 = 1$, $w_3 = 1$, and $w_4 = 1$. Ray transitions are one indicator for the rendering performance of a distributed path tracing system. The image shows an assignment of the PBRT Ecosys and the Moana Mountain models to 16 nodes. We can see from the images that the number of ray transitions is comparable or even lower for the combined heuristic than for the two simpler heuristics. Note however that the weight assignment (and the heuristic itself) are not fine-tuned for rendering performance but for memory efficiency.*

construction schemes for our algorithm where the individual nodes would not need to have access to the actual geometry, but only to the instances and the precomputed SVT. We believe that the construction scheme based on first projecting the triangle vertices on a uniform grid, and then using an SVT to build a spatial index, might even be useful for in-core ray tracing. This is also a direction we consider to further investigate in the future. Our evaluation so far has concentrated on memory consumption and only assumed that the spatial distribution will have an influence on the rendering performance. In the future we intend to integrate our algorithm in a distributed, data-parallel rendering pipeline and test this assumption.

Acknowledgments

The authors wish to thank the people who make the 3-d models for our evaluation available. The Moana Island Scene was published by Rasmus Tamstorf and Heather Pritchett from Wald Disney Animation Studios. The other models are maintained online as part of the PBRT scene repository [PJH]. The PBRT Ecosys data set was originally presented in [DHL*98]. The PBRT Landscape data set is provided by Tim Dapper under the CC BY 4.0 license (https://creativecommons.org/licenses/by/4.0/). All 3-d models are available in `pbrt` format.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 145–149. URL: http://doi.acm.org/10.1145/1572769.1572792, doi:10.1145/1572769.1572792. 4

[ANG*18] ABRAM G., NAVRÁTIL P., GROSSETT P., ROGERS D., AHRENS J.: Galaxy: Asynchronous ray tracing for large high-fidelity visualization. In *The 8th IEEE Symposium on Large Data Analysis and Visualization* (2018), IEEE. 3

[BLZD12] BAO G., LI H., ZHANG X., DONG W.: Large-scale forest rendering: Real-time, realistic, and progressive. *Computers & Graphics 36*, 3 (2012), 140 – 151. Novel Applications of VR. doi:https://doi.org/10.1016/j.cag.2012.01.005. 3

[BPR02] BARTZ D., PUEYO X., REINHARD E.: "kilauea"–parallel global illumination renderer. 2

[Car05] CARUCCI F.: Inside geometry instancing. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (GPU Gems)*, Pharr M., Randima F., (Eds.). Addison Wesley, 2005, ch. 3, pp. 47–68. 3

[CFS*18] CHRISTENSEN P., FONG J., SHADE J., WOOTEN W., SCHUBERT B., KENSLER A., FRIEDMAN S., KILPATRICK C., RAMSHAW C., BANNISTER M., ET AL.: Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics (TOG) 37*, 3 (2018), 1–21. 2

[DCSD02] DEUSSEN O., COLDITZ C., STAMMINGER M., DRETTAKIS G.: Interactive visualization of complex plant ecosystems. In *IEEE Visualization, 2002. VIS 2002.* (Oct 2002), pp. 219–226. doi:10.1109/VISUAL.2002.1183778. 3

[DHL*98] DEUSSEN O., HANRAHAN P., LINTERMANN B., MĚCH R., PHARR M., PRUSINKIEWICZ P.: Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 275–286. URL: http://doi.acm.org/10.1145/280814.280898, doi:10.1145/280814.280898. 10

[GHFB13] GU Y., HE Y., FATAHALIAN K., BLELLOCH G.: Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 81–88. URL: http://doi.acm.org/10.1145/2492045.2492054, doi:10.1145/2492045.2492054. 4

[GIF*18] GEORGIEV I., IZE T., FARNSWORTH M., MONTOYA-VOZMEDIANO R., KING A., LOMMEL B. V., JIMENEZ A., ANSON O., OGAKI S., JOHNSTON E., HERUBEL A., RUSSELL D., SERVANT F., FAJARDO M.: Arnold: A brute-force production path tracer. *ACM Trans. Graph. 37*, 3 (Aug. 2018), 32:1–32:12. URL: http://doi.acm.org/10.1145/3182160, doi:10.1145/3182160. 3

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (May 1987), 14–20. doi:10.1109/MCG.1987.276983. 4

[Joh09] JOHANSSON M.: Integrating occlusion culling and hardware instancing for efficient real-time rendering of building information models. In *GRAPP/IVAPP* (2009). 3

[KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. In *Proceedings of the Twenty-second Eurographics Conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2011), EGSR '11, Eurographics Association, pp. 1353–1360. URL: http://dx.doi.org/10.1111/j.1467-8659.2011.01995.x, doi:10.1111/j.1467-8659.2011.01995.x. 3

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum* (2009). doi:10.1111/j.1467-8659.2009.01377.x. 4

[MB18] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 03 (Mar 2018), 1345–1353. doi:10.1109/TVCG.2017.2669983. 4

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl. 14*, 4 (July 1994), 23–32. URL: https://doi.org/10.1109/38.291528, doi:10.1109/38.291528. 3

[ME17] MORRICAL N., EDWARDS J.: Parallel quadtree construction on collections of objects. *Computers & Graphics 66* (2017), 162–168. 4

[MM10] MARCHESIN S., MA K.-L.: Cross-node occlusion in sort-last volume rendering. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2010), EGPGV'10, Eurographics Association, pp. 11–18. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV10/011-018, doi:10.2312/EGPGV/EGPGV10/011-018. 3

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.-M.: Dynamic load balancing for parallel volume rendering. In *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2006), EGPGV '06, Eurographics Association, pp. 43–50. URL: http://dx.doi.org/10.2312/EGPGV/EGPGV06/043-050, doi:10.2312/EGPGV/EGPGV06/043-050. 3

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), Heirich A., Raffin B., dos Santos L. P., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV06/059-066. 3

[NCFL14] NAVRÁTIL P. A., CHILDS H., FUSSELL D. S., LIN C.: Exploring the spectrum of dynamic scheduling algorithms for scalable distributed-memory ray tracing. *IEEE Transactions on Visualization and Computer Graphics 20*, 6 (June 2014), 893–906. doi:10.1109/TVCG.2013.261. 3

[NFLC12] NAVRÁTIL P. A., FUSSELL D. S., LIN C., CHILDS H.: Dynamic Scheduling for Large-Scale Distributed-Memory Ray Tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), Childs H., Kuhlen T., Marton F., (Eds.), The Eurographics Association. doi:10.2312/EGPGV/EGPGV12/061-070. 3

[PJH] PHARR M., JAKOB W., HUMPHREYS G.: Scenes for pbrt-v3. https://www.pbrt.org/scenes-v3.html. Accessed: 2019-06-19. 10

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory To Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016. 3, 4

[SDF*13] SLOMP M., DARIBO I., FURUKAWA R., SAGAWA R., HIURA S., ASADA N., KAWASAKI H.: Hardware-accelerated geometry instancing for surfel and voxel rendering of scanned 4d media. In *11th International Conference on Quality Control by Artificial Vision (QCAV)* (2013). 3

[SF14] SANTOS P. I. N., FILHO W. C.: Instanced rendering of massive CAD models using shape matching. In *2014 27th SIBGRAPI Conference on Graphics, Patterns and Images* (Aug 2014), pp. 335–342. doi:10.1109/SIBGRAPI.2014.34. 3

[SY17] SON M., YOON S.-E.: Timeline scheduling for out-of-core ray batching. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, ACM, pp. 11:1–11:10. URL: http://doi.acm.org/10.1145/3105762.3105784, doi:10.1145/3105762.3105784. 3

[TP18] TAMSTORF R., PRITCHETT H.: Moana Island Scene (v1.1) [Data set]. https://www.technology.disneyanimation.com/islandscene, 2018. Walt Disney Animation Studios. 1, 2

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools 13*, 2 (2008), 21–36. 6

[Wal] WALD I.: pbrtParser – Parser for PBRT files. https://gitlab.com/ingowald/pbrt-parser. Accessed: 2020-03-03. 7

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 3

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An interactive out-of-core rendering framework for visualizing massively complex models. In *Eurographics Workshop on Rendering* (2004), Keller A., Jensen H. W., (Eds.), The Eurographics Association. doi:10.2312/EGWR/EGSR04/081-092. 3

[WH06] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *IEEE Symposium on Interactive Ray Tracing 2006(RT)* (09 2006), vol. 00, pp. 61–69. URL: doi.ieeecomputersociety.org/10.1109/RT.2006.280216, doi:10.1109/RT.2006.280216. 4

[WLT13] WANG S., LIN C., TAI W.: Compressing 3d trees with rendering efficiency based on differential data. *IEEE Transactions on Multimedia 15*, 2 (Feb 2013), 304–315. doi:10.1109/TMM.2012.2231062. 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. URL: http://doi.acm.org/10.1145/2601097.2601199, doi:10.1145/2601097.2601199. 3

[ZSL18] ZELLMANN S., SCHULZE J. P., LANG U.: Rapid k-d tree construction for sparse volume data. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), Childs H., Cucchietti F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20181097. 6

[ZWL17] ZELLMANN S., WICKEROTH D., LANG U.: Visionaray: A cross-platform ray tracing template library. In *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)* (2017), IEEE. 7