The Visual Computer (2023) 39:3797–3809
https://doi.org/10.1007/s00371-023-02975-y

ORIGINAL ARTICLE

A neural builder for spatial subdivision hierarchies

Iordanis Evangelou[1] · Georgios Papaioannou[1] · Konstantinos Vardis[1] · Anastasios Gkaravelis[1]

Accepted: 14 June 2023 / Published online: 24 July 2023
© The Author(s) 2023

Abstract

Spatial data structures, such as k-d trees and bounding volume hierarchies, are extensively used in computer graphics for the acceleration of spatial queries in ray tracing, nearest neighbour searches and other tasks. Typically, the splitting strategy employed during the construction of such structures is based on the greedy evaluation of a predefined objective function, resulting in a less than optimal subdivision scheme. In this work, for the first time, we propose the use of unsupervised deep learning to infer the structure of a fixed-depth k-d tree from a constant, subsampled set of the input primitives, based on the recursive evaluation of the cost function at hand. This results in high-quality upper spatial hierarchy, inferred in constant time and without paying the intractable price of a fully recursive tree optimisation. The resulting fixed-depth tree can then be further expanded, in parallel, into either a full k-d tree or transformed into a bounding volume hierarchy, with any known conventional tree builder. The approach is generic enough to accommodate different cost functions, such as the popular surface area and volume heuristics. We experimentally validate that the resulting hierarchies have competitive traversal performance with respect to established tree builders, while maintaining minimal overhead in construction times.

Keywords Rendering · Neural networks · Spatial hierarchies · Ray tracing

1 Introduction

Spatial data structures are extensively used in computer graphics for the acceleration of spatial queries, with the most notable applications being ray-geometry intersection tests and nearest neighbour search in point clouds. Typically, a hierarchical spatial data structure of a bounded spatial domain, containing a population of primitives, follows one of two strategies: spatial subdivision and bounding volume hierarchies (*BVH*). The first, mainly represented by the k-d tree and octree data structures, follows a spatial decomposition methodology and ensures that no overlapping bounding volumes are encountered among siblings, but does not guarantee a tight enclosing of the contained primitives at each node. On the other hand, BVH subdomain bounds can be—and usually are—fitted on the corresponding subset of primitives, but cannot guarantee the non-overlap-ping property. However, both strategies share a common problem: determining the optimal bound hierarchy is not a trivial task. Current state-of-the-art builders have a high build and traversal performance, but typically rely on the local tree cost evaluation, resulting in nearly optimal hierarchies. Exploiting a fully recursive tree cost evaluation would yield an optimal tree, but the evaluation itself is intractable, even for shallow trees and small inputs.

In this paper, for the first time, we exploit the generality of *unsupervised deep learning* to infer the structure of a k-d tree, based on the recursive evaluation of the cost function at hand, including the very popular surface area heuristic (*SAH*). The immediate benefits from addressing the hierarchy construction from a deep learning standpoint are: (a) the decoupling of hierarchy construction time from the primitive count, leading to constant hierarchy inference times, (b) compatibility with any linear recursive cost function (see Eq. 4) and (c) the mapping of the construction process to efficient generic hardware (tensor cores). As shown later in the paper, the explicit use of arbitrary recursive cost functions in our model, result

✉ Iordanis Evangelou
iordanise@aueb.gr

Georgios Papaioannou
gepap@aueb.gr

Konstantinos Vardis
kvardis@aueb.gr

Anastasios Gkaravelis
agkar@aueb.gr

[1] Department of Informatics, Athens University of Economics and Business, Athens, Greece

in highly optimised hierarchies without paying the respective (intractable) cost.

At a glance, the proposed neural builder produces a high-quality spatial partitioning of the top levels of the hierarchy of an input scene. During the inference (tree construction) step, the input of the network is a subsampled, fixed set of input primitives, that decouples the scene size from the number of network parameters. Then, the architecture determines the splits by exploring all possible path permutations. The output of the network is a fixed-depth k-d tree. This tree is then trivially transformed and expanded into a complete acceleration data structure (of the same node layout), using any conventional builder and whose exact form is driven by the needs of the application at hand. The disjoint clusters stemming from the neural hierarchy, allow for trivial data parallelism in both CPU and GPU architectures in order to expand its subtrees. We experimentally validate the hybrid building process with two prominent cases (see Sect. 6): nearest neighbour queries, by expanding the subtrees into a traditional k-d tree and ray-primitive intersection tests, by transforming the upper tree into a BVH and expanding (in parallel) the resulting leaves.

It is important to note here, that our approach directly optimises the tree cost, rather than the tree structure itself, e.g. as in similarity-based inference, since the latter would be problematic. First, similarity-based inference would depend on the evaluation of the final loss against reference trees in a supervised manner, requiring the definition of a robust and tolerant distance metric between hierarchical data structures. We avoid this, since this is an ill-formed problem. More importantly, employing such an approach, directly assumes that a unique cost-minimising tree configuration exists for every input, which is probably not true, in the general case. Second, although the algorithm to compute the recursive cost function is straightforward, it fits poorly to the gradient descent paradigm, at least in an intuitive way. In this paper we provide the necessary details to make this mapping possible.

2 Preliminaries

In this paper we focus on the construction of k-d trees and in particular, 3-d trees—which we further transform, as necessary, although the methodology is easily generalized. A k-d tree is a binary spatial partitioning hierarchy, whose internal nodes act as axis-aligned hyper-planes that decompose the node's bounding volume into two non-overlapping subdomains.

The construction of a k-d-tree takes an input a set of points $\mathbf{P} \subset \mathbb{R}^3$ and recursively subdivides them until a certain condition is met. In the following, all points and local bounds at each node to be split are expressed in the normalized root node space, hence, $\mathbf{P} \subset [0, 1]^3$. Construction begins from the root node $\mathcal{P} = (\mathbf{P}, \mathbf{B})$, splitting the bounding box $\mathbf{B} = (\boldsymbol{b}_{\min}, \boldsymbol{b}_{\max})$ into two child nodes $\mathcal{P}_L$ and $\mathcal{P}_R$, by first choosing a plane with an axis-aligned normal $\mathbf{n} \in \{\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z\}$

and offset $b \in [b_{\min}, b_{\max}]$. Child nodes are initialized with the mutually exclusive point subsets $\mathbf{P}_L$ and $\mathbf{P}_R$ contained in the non-overlapping partitions:

$$\mathbf{B}_L = g_L(\mathbf{n}, b, \mathbf{B}) = (\mathbf{b}_{\min}, (\mathbf{1} - \mathbf{n}) \odot \mathbf{b}_{\max} + b\mathbf{n}) \quad (1)$$

$$\mathbf{B}_R = g_R(\mathbf{n}, b, \mathbf{B}) = ((\mathbf{1} - \mathbf{n}) \odot \mathbf{b}_{\min} + b\mathbf{n}, \mathbf{b}_{\max}), \quad (2)$$

where $\odot$ denotes component-wise vector multiplication.

Note that, in general, the tight bound of the points in each subspace may be smaller than the corresponding subdivision of $\mathcal{P}$. If a transformation of a k-d tree into a BVH is required (Sect. 6.4), tight bounds must be calculated during the conversion process.

In order to build an acceleration data structure that is efficient to traverse, the quality of a tree is typically measured using an objective function C and an optimisation algorithm decides whether to stop exploring alternative topologies. Candidate configurations for a node $\mathcal{P}$ are directly drawn from the composite domain $\mathcal{D} = \mathcal{D}_x \cup \mathcal{D}_y \cup \mathcal{D}_z$ that contains every possible pair of discrete axis-aligned splits and continuous offsets:

$$\mathcal{D}_x = [\mathbf{n}_x, b], \quad \mathcal{D}_y = [\mathbf{n}_y, b],$$
$$\mathcal{D}_z = [\mathbf{n}_z, b], \quad b \in [b_{\min}, b_{\max}]. \quad (3)$$

Child nodes are recursively generated with bounds given in Eqs. 1 and 2. A general recursive form for the tree traversal cost, that encompasses most practical cost functions encountered in computer graphics algorithms, is:

$$C(\mathcal{P}) = \min_{\mathcal{D}} \begin{cases} p(\mathcal{P}) + w_L(\mathcal{P})C(\mathcal{P}_L) + w_R(\mathcal{P})C(\mathcal{P}_R) & \text{internal} \\ q(\mathcal{P}) & \text{leaf} \end{cases}, \quad (4)$$

where $w_{L/R}(\mathcal{P})$ is a weighting function for each branch, $q(\mathcal{P})$ is the cost of visiting a leaf and $p(\mathcal{P})$ is the traversal cost for an internal node. A widely used cost function for the optimisation of spatial hierarchies and particularly for ray tracing, is the *Surface Area Heuristic* [8, 20]. Its mapping to Eq. 4 is:

$$w_{L/R}(\mathcal{P}) = \frac{SA(\mathbf{B}_{L/R})}{SA(\mathbf{B})}, \quad p(\mathcal{P}) = c_t,$$
$$q(\mathcal{P}) = c_i N(\mathcal{P}), \quad (5)$$

where $SA(\mathbf{B})$ is the surface area of bounding box $\mathbf{B}$, $N(\mathcal{P})$ is the number of primitives in $\mathcal{P}$ and c_t, c_i are constant penalty

factors for traversing a node and intersecting a primitive, respectively. The goal of this objective is to reduce the cost of non-terminating rays traversing the constructed hierarchy by measuring the probability of hitting each node through Eq. 5.

Similar to the SAH, the *Voxel Volume Heuristic* (VVH) [31] is a cost function employed in acceleration data structures for efficient nearest neighbour queries, mapped to Eq. 4 as:

$$
\begin{aligned}
w_{L/R}(\mathcal{P}) &= \frac{V(\mathbf{B}_{L/R})}{V(\mathbf{B})}, \\
p(\mathcal{P}) &= c_t, \quad q(\mathcal{P}) = c_i N(\mathcal{P}),
\end{aligned}
\tag{6}
$$

where $V(\mathbf{B})$ is the corresponding volume of bound $\mathbf{B}$, expanded by a small offset in all directions. This offset typically corresponds to the radius search as in the photon mapping [13] algorithm, but may tend to zero for general neighbourhood estimation.

From an algorithmic perspective, the process of subdivision selects all possible splits at each level, evaluates the cost and then recursively repeats this process until the leaf condition is met. The final cost can be aggregated upwards from the leaves, through each hierarchy path and the minimum-cost path is stored along the way.

It is worth clarifying here is that, while the k-d builder is an inherently top-down *recursive* builder, the cost function evaluation may not be. The cost function can be either locally or globally minimised, the former case resulting in a greedy cost evaluation and the latter leading to the global optimum. The greedy, local cost evaluation is adopted in practical implementations, since the complexity of the fully recursive cost function evaluation is computationally impractical.

3 Related work

Spatial data structures. Many algorithms have been proposed for the construction of hierarchical spatial structures via the minimisation of a cost function. Balancing between construction time and final tree quality/traversal performance can be notoriously difficult. The widely used approaches are categorised as either spatial subdivision methods or object-based strategies.

Spatial decomposition into non-overlapping domains is typically performed via axis-aligned domain splits. The most popular space partitioning scheme and indexing structure is the k-d tree [1], for which both CPU and GPU algorithms [31, 32, 36] exist, optimising the greedy SAH and VVH cost functions, to accelerate ray traversal and radius search queries, respectively. A notable alternative is the use of shallow and wide regular [14] or irregular [27] grids, optimised with a greedy variant of SAH and targeting fast index construction.

Hierarchical clustering for nonzero area primitives, such as triangles, can also be done with object-based subdivision. Contrary to spatial subdivision, the notable property guaranteed here is the unique correspondence between primitives and nodes. Node boundaries correspond to tight primitive cluster extents, maximising empty space. The Bounding Volume Hierarchy adopts this strategy and has seen a wide attraction in the literature, especially for the ray tracing task. Several approaches have been proposed for BVH construction, based on the SAH heuristic and most notably for the greedy case [29, 30]. Specialised optimisation algorithms can also be employed as a post-processing step to further improve the quality of an already constructed BVH [6, 15, 23], or guide the construction from auxiliary hierarchies [7, 10, 12]. For a detailed analysis on relevant data structures, the interested reader is referred to the recent survey by [24].

Point-based learning. Here we briefly summarize several neural architectures that have been proposed for learning representations of unstructured point sets. Most notably, the PointNet architecture [4] has set the baseline for shape classification and segmentation tasks, while the theoretical foundation of robust point learning was established with Deep Sets [35], where the authors have rigorously defined and applied permutation-invariant and equivariant layers for feature extraction.

Point-based learning has also demonstrated its expressive power with generative models for 3D shape generation and reconstruction. Most recent approaches either rely on shape priors or exploit the input set as a prior to generate the actual mesh, in the SP-GAN [18] and Point2Mesh [9] architectures, respectively.

Our work diverges from the above literature, investigating a different geometric task; we process input point sets and learn a hidden representation that optimises an analytic form of a recursive cost function to infer a spatial subdivision data structure.

Structure-aware learning. Several learning-based approaches operate hierarchically to exploit the local structure of the data representation. Although these methods originate from a different application domain, address different tasks and are orthogonal to this work, we briefly discuss them in order to clearly distinguish them from our contributions.

Fixed k-d trees, with trainable layers as nodes, have been shown to assist in the agglomeration of features for shape recognition, retrieval and part-based segmentation [17].

Similarly, uniform grid [21] and Octree [28] hierarchies also exploit spatial subdivision for hierarchical feature extraction. While these structures are statically constructed, adaptive Octrees [34] were also proposed for 3D shape encoding, to handle issues related to learning effectiveness and resource demands. They use learnable features to decide on whether to split an octree node or not, up to a fixed depth.

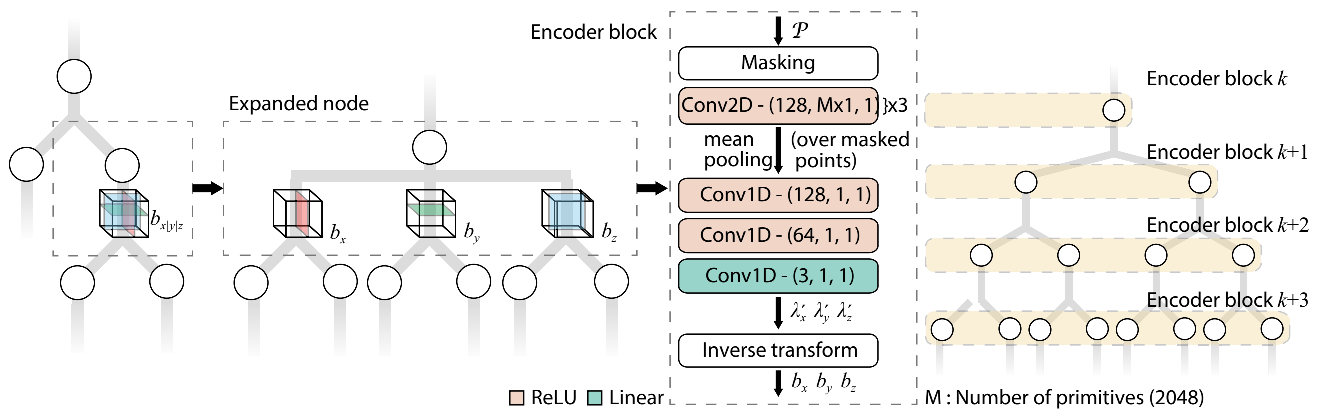

Fig. 1 The mapping of a binary spatial subdivision hierarchy to our neural network architecture. From left to right: an arbitrary axis-aligned split is expanded as three individual ones and the resulting 6-way branching node is mapped to a neural encoder block. Each encoder is re-used across the nodes of the same tree level

On the contrary, in this work we focus on building a model that implicitly determines the hierarchy, through minimisation of a recursive and task-specific cost function.

Last but not least, the pioneering graph processing methods using neural networks [5, 16], operate directly on input nodes of fixed size, extract individual features and aggregate them for graph classification or link prediction. This is contrary to the problem at hand, where disjoint clusters must be discovered and hierarchically structured.

4 The neural *k*-d tree builder

A key observation in order to map the splitting hyperplane selection to a neural network architecture is that the hyperplane normal is discretised. As a result, we can remove the normal selection from the original domain $\mathcal{D}$ by expanding each binary node into three separate tuples $\mathcal{P}$, one for each candidate splitting axis, and simultaneously evaluating the cost for all of them (Fig. 1-left), similar to a pooling operator, across all branches. The optimisation process is effectively reduced to only determining the best splitting hyperplane offsets b_j, simultaneously over all axial split candidate subtrees, so that the tree cost C is minimised. For a formed tree $\mathcal{T}^{(k)}$ of depth k, we need to minimise the loss $L(C; \boldsymbol{\theta})$ over the predicted recursive tree cost and ensure the proper differentiation of the loss over trainable parameters $\boldsymbol{\theta}$, involving gradient propagation through the cost Eq. 4.

4.1 Architecture

As illustrated in Fig. 1, the k-d tree is modelled as a set of cascaded encoders, each encoder representing a binary tree node expanded into a hex node, encompassing all axial splits. Each encoder receives the data $\mathcal{P}$ corresponding to the current spatial partition and outputs the offsets of the three potential splits, in a top-down, level-order fashion. Since nodes belonging to the same tree level tend to be related in terms of shape granularity, an encoder is shared among all nodes in the same level (Fig. 1-right). This enforces invariance of the encoders with respect to absolute position in the input domain and the number of points in the subspace. To this end, translation and scale are removed from the encoder's representation, due to prior normalisation of the encoder's input point set with respect to its tight bounds. Since the same input is propagated down the tree and distributed (due to the appropriate masks) across the nodes of a tree level, the dimensions of the encoder's latent representation and, consequently, their descriptive capability, remains the same, despite gradual expansion of the tree. Not surprisingly, sharing the training parameters across the entire tree is not as effective, since the distribution of primitives varies significantly across levels, in general, thus requiring a far larger network capacity.

Input sampling. Realistic applications tend to operate on prohibitively large inputs for current neural network architectures. To reduce the input size, we uniformly draw a fixed number of M samples from the original population, preserving the spatial distribution of the primitives. We experimentally validated that a small, subsampled version of the actual input is sufficient to retain the high quality of the tree, up to a certain depth. In our implementation, 2048 samples are used. For more details, see Sect. 6.2.

It is important to note that subsampling is applicable only because we estimate a spatial decomposition of non-overlapping subdomains (a k-d tree here); the node bounds are not directly computed from the contained points, so the final hierarchy can be first computed and then populated with the original primitives. This is in sharp contrast to object-

based subdivisions, where topology is invalidated when the input is modified.

The encoder represents a tree node and receives a fixed set of M points, normalised with respect to the bounding box of the input partition, and shifted by one unit so that coordinates are in the $[1, 2]$ interval. This way, zero is used for the representation of masked out points that do not reside within the bounds $\mathbf{B}$ of the current space partition, acting as an attention mechanism. When the input points are propagated down through the cascaded encoders, we check them against the bounding box $\mathbf{B}$ of the current partition to generate a mask, which retains or zeroes the respective point coordinates.

The output of the encoder is a vector $\boldsymbol{\lambda}'$ containing the split offsets along each axis, normalised with respect to the *tight bounds* of the active points in the current tree node. Through a linear transformation (denoted as Inverse transform in Fig. 1), these offsets are first converted to offsets $\boldsymbol{\lambda} = (\lambda_x, \lambda_y, \lambda_z)$ within the partition bounding box and subsequently to root-level offsets:

$$\boldsymbol{b} = f_b(\mathbf{n}, \boldsymbol{\lambda}, \mathbf{B}) = \mathbf{n} \odot [\mathbf{b}_{\min} + \boldsymbol{\lambda}(\mathbf{b}_{\max} - \mathbf{b}_{\min})]. \tag{7}$$

Predicting the split offsets with respect to the tight point set bounds is important. First, the input of the encoder must be bound-agnostic, to allow the encoder block parameters to be shared among nodes of the same level but also for the input to be cascaded down the tree using identical encoder modules. Second, as explained in Sect. 5.2, the change of variable from the global reference frame to the tight bounds of each partition solves a serious differentiability issue, affecting the partial derivatives of lower bounds with respect to splitting offsets of upper bounds. Last but not least, it makes easier to enforce valid parameters for split offsets, during optimisation, as will be discussed next, in Sect. 4.2.

Our encoder consists of the layers presented in the Encoder block of Fig. 1, resulting in 58113 trainable parameters. In the above encoding scheme, the six-way branching of each node and thus the expanded, simultaneous and independent evaluation of the cost effectively reduces our task to a one-dimensional problem and decouples the splitting decision from the elements of the other dimensions. This means that we can share the training parameters across every dimension using standard convolutional layers, reduce the domain dimensionality further and, most importantly, implicitly force generalisation of the encoder to handle axial permutations of the input coordinates.

4.2 Training

During the training phase, our model consumes batches of M normalised points. The augmented tree is constructed in a top-down, level-order manner, determining the offsets b_i for spatial subdivision in each node. Then, we agglomeratively traverse and apply the cost function (Eq. 4) in a bottom-up, level-order fashion to find the cost of the minimum tree and consequently, minimise the loss over it. Since this is an unsupervised method, the loss function embodies the minimisation of its norm.

Loss function. In our model we experimentally found the $\ell 2$ norm to achieve the best quality:

$$L_{\text{tree}}(C) = \frac{1}{|\mathcal{B}|} \sum_{i=0}^{|\mathcal{B}|} \left|\left|\tilde{C}_i\right|\right|_2^2, \tag{8}$$

where $\mathcal{B}$ is the batch and $\tilde{C}$ is the normalised optimal tree cost relative to the cost of being a single root node.

Additionally, we force the local split offset $\boldsymbol{\lambda}'$ to gracefully remain within the normalised bound with the following, differentiable penalty function:

$$L_{\text{b}}(\lambda') = \mathbb{I}_{x<0}(x)\ell 1_{\text{soft}}(x) + \mathbb{I}_{x>1}(x)\ell 1_{\text{soft}}(1-x) \tag{9}$$

where $\ell 1_{\text{soft}}(\cdot)$ is the Huber loss [11], with parameter 0.1. The penalty over the entire tree is then:

$$L_{\text{bound}}(\mathcal{T}) = \frac{1}{|\mathcal{B}|} \sum_{i=0}^{|\mathcal{B}|} \sum_{j=0}^{|\mathcal{T}^{(k)}|} \gamma \left(L_{\text{b}}(\lambda'_{x_{ij}}) + L_{\text{b}}(\lambda'_{y_{ij}}) + L_{\text{b}}(\lambda'_{z_{ij}}) \right), \tag{10}$$

where γ, is a positively defined function, controlling the relative contribution of the splitting offset constraint. Intuitively, the higher in the hierarchy the split is, the larger the subtree that an out-of-tight-bounds offset invalidates. Therefore for a tree of depth d, we set γ proportional to the number of descendant internal nodes from the current level k: $\gamma = 2^{d-k} - 1$. Finally, the total loss is the linear combination of the two components:

$$L(C, \mathcal{T}; \theta) = L_{\text{tree}}(C) + L_{\text{bound}}(\mathcal{T}). \tag{11}$$

Gradients. Proper updates of the trainable parameters $\boldsymbol{\theta}$ during backpropagation, introduce two fundamental design properties that must be taken into account. First, gradients should only be accumulated from offset predictions that actually participate in paths associated with the currently minimum tree cost and the rest must be disregarded. Second, since we are recursively evaluating the cost function, we are imposing the dependence of each node to the decision of preceding nodes, up to the root. The latter implies that proper gradients should exist and be accumulated from every child node in the minimum tree, requiring special handling of specific components, as well as proper scaling of the

gradients due to normalisation of the input in each cascaded encoder.

The calculation of gradients for the encoder is discussed in Sect. 5. We devote an entire section to this, since our contributions to enable the differentiability of the recursive cost function are crucial for the determination of a spatial subdivision with a gradient-based optimiser.

4.3 Tree inference

In our method, the actual hierarchy does not explicitly participate in the loss function, but is rather optimised implicitly through its cost. The inference of the hierarchy can be performed either recursively or greedily and expanded with any traditional builder.

Recursive inference. Similar to the training phase, we apply a forward pass that infers every path permutation and then, starting from the leaves, we agglomeratively apply Eq. 4 in a level-order fashion. When a path that participated on the minimum tree is decided, we append the plane associated with that cost. The process is terminated, when the root node is evaluated.

Greedy inference. A more resource-efficient way to infer the final tree is to follow a greedy splitting strategy. This involves only a top-down, level-order traversal over the inferred tree. For each $\mathcal{P}$, we can evaluate the local cost using Eq. 4 and decide whether to place a split and continue subdividing until the maximum trained depth is reached or, stop and declare $\mathcal{P}$ as a leaf. Please keep in mind that the offsets at each node have been already estimated with the recursive cost evaluation. Only the split axis selection is greedily estimated here, based on that recorded cost. Therefore, in contrast to a greedy builder, the solution provided by the greedy inference step uses local decisions albeit for a globally optimised cost.

4.4 Hierarchy population

When the final structure is emitted, it must be populated with the entire set of primitives. We exploit the fact that in a non-overlapping spatial domain decomposition of a domain **B**, the resulting hierarchy remains valid without any modifications, for arbitrary subsets of the contained primitives in **B**. Hence, after the inference of the splits using the subsampled input, the application should only distribute the full set of primitives to the leaf nodes and re-evaluate Eq. 4 to get the final cost.

If adaptation of the leaf bounds is required, e.g. in order to reform the tree as a BVH (Sect. 6.4), the hierarchy can be refitted in a single linear pass, since the child-to-parent connections are already inferred from the network.

4.5 Hierarchy expansion

Starting from the clustering produced by the leaves, any existing algorithm can be employed to extend the hierarchy. This makes our method easy to integrate and combine with any existing builder, as demonstrated in our application case studies (Sects. 6.3 and 6.4). It is important to note that having generated a low-cost top part of the hierarchy, its subtrees can be independently computed in a trivially parallel manner.

5 Differentiability of the cost function

Here we provide details about the end-to-end differentiability of the builder. Using Eqs. 8 and 4, the partial derivative of our objective loss function over a trainable parameter are:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial C}\frac{\partial C}{\partial \theta_i}. \tag{12}$$

While the first partial is trivial, our work contributes to the evaluation of the second partial derivative.

5.1 Gradients for splitting plane selection

At each tree node, during a forward pass through the tree, the cost evaluation uses the minimum of the three separate axial split costs. To be able to propagate gradients through this three-way selector during back propagation, we do the following.

First, for a formed binary tree $\mathcal{T}^{(k)}$ of depth k, we expand each node into three separate tuples $\mathcal{P}$, one for each candidate splitting axis, so that we can now simultaneously evaluate the cost for all of them (Fig. 1-left), similar to the pooling operator. The number of nodes participating in the candidate path formation becomes $|\mathcal{T}^{(k)}| = (6^k - 1)/5$ instead of $2^k - 1$ for the binary tree. Now, the trainable parameters only affect the splitting plane offsets b_j and the gradient of the cost for an arbitrary parameter θ_i becomes:

$$\frac{\partial C}{\partial \theta_i} = \sum_{j=0}^{|\mathcal{T}^{(k)}|} \frac{\partial C}{\partial b_j}\frac{\partial b_j}{\partial \theta_i}. \tag{13}$$

By exploiting both properties of axis-independence and parameter sharing discussed in Sect. 4.1, we can now adapt Eq. 13 to obtain the partial derivative of the path selection using the negative $LogSumExp$ function $nLSE(\cdot;t)$ transformation during the backward pass, where parameter t controls the sharpness of the approximation (here always $t = 1$). This particular function has the nice property that its gradient is the $softmax$ function, which forces gradients

to propagate through the path with minimum cost and suppressing the rest in a continuous and smooth manner.

It is nevertheless important to note that the agglomerative pooling operator over the neural nodes must be differentiable. This is in sharp contrast to the usual approaches adopted in convolutional networks, which apply non-differentiable forms of pooling. The need arises when more than one paths through an internal node evaluate to the same cost; due to the recursive cost evaluation, arbitrarily choosing one of them will lead to dropping the update of subtree configurations through gradient back-propagation, which should normally be updated.

5.2 Differentiating bound selection

When a split is followed at any level by a split in the same axis (e.g. $\mathbf{n}_x$ followed by $\mathbf{n}_y$ and then by $\mathbf{n}_x$), the partial derivatives of lower bounds with respect to splitting offsets of upper bounds are zero. To overcome this, Eqs. 1 and 2 are re-parameterised, so that the split offset is evaluated as a linear combination with factor $\boldsymbol{\lambda}$ of the partition's bounding box corners:

$$\boldsymbol{b} = f_b(\mathbf{n}, \boldsymbol{\lambda}, \mathbf{B}) = \mathbf{n} \odot [\mathbf{b}_{\min} + \boldsymbol{\lambda}(\mathbf{b}_{\max} - \mathbf{b}_{\min})]. \tag{14}$$

The child node bounding boxes $\mathbf{B}_L, \mathbf{B}_R$ are now updated according to the modified functions of Eqs. 1 and 2:

$$g_L(\mathbf{n}, \boldsymbol{\lambda}, \mathbf{B}) = (\boldsymbol{b}_{\min}, (\mathbf{1} - \mathbf{n}) \odot \boldsymbol{b}_{\max} + f_b(\boldsymbol{\lambda}, \mathbf{B})) \tag{15}$$

$$g_R(\mathbf{n}, \boldsymbol{\lambda}, \mathbf{B}) = ((\mathbf{1} - \mathbf{n}) \odot \boldsymbol{b}_{\min} + f_b(\boldsymbol{\lambda}, \mathbf{B}), \boldsymbol{b}_{\max}). \tag{16}$$

The vector of interpolation parameters $\boldsymbol{\lambda}^{(m)}$ is the output of an encoder, operating on the point cloud at any node m through learnable parameters θ_i (see Fig. 1-middle).

In order to make the encoder agnostic to bound extents and therefore be able to generalise its use throughout the data structure, as discussed in Sect. 4.1, instead of using the offset λ defined above, the encoder computes and outputs an offset λ' for the splitting plane normalised with respect to the tight bounds of the given input points. This is why the inverse mapping of λ' to global node bound space is necessary at the end of the encoder (inverse transform in Fig. 1-middle).

5.3 Differentiating primitive count functions

Frequently, the constituent parts of the cost function depend on the number of points $N(\mathcal{P})$ included in the two subdomains of a split node:

$$N_{L/R}(b; \mathcal{P}) = \sum_{p \in \mathbf{P}} \mathbb{I}_{p \in \mathbf{P}_{L/R}}(b). \tag{17}$$

This is a piece-wise constant function with respect to the split offset, which, although trivial to implement algorithmically, results in zero gradient almost everywhere. To avoid such an undesirable behaviour in a gradient-based optimisation process, we explicitly provide a replacement gradient, which guides the optimiser out of the plateau of the piece-wise constant function.

While in theory a suitable gradient replacement could result from a linear combination of sigmoid functions, in practice, this would be problematic, since careful parameterisation of the sigmoids is required individually for each step in the query domain around the splitting offset to avoid vanishing gradients (wide step) or exploding gradients (narrow step). In the following, we propose an alternative smooth replacement function for gradient computation, which avoids these issues.

We only discuss here the case of counting the elements for the left half-space, since the analysis for the other subdomain is symmetrical. Given Eq. 17 and an arbitrary point b_0 in the valid domain we are interested in efficiently determining the smallest step ϵ such that it minimally increments the total count in the space partition by n:

$$N(b_0 + \epsilon; \mathcal{P}) = N(b_0; \mathcal{P}) + n, \tag{18}$$

or equivalently, at any point b_0 we are interested in the gradient that minimally increments the total count. Requiring that the function's gradient is additionally bounded, we can define a replacement gradient for the step-wise function at b_0: $\partial N/\partial b = n/(\epsilon + e)$, where e is a small positive constant. The new gradient is continuous almost everywhere (except at the finite step discontinuities) and bounded.

In the backward pass, the evaluation of ϵ at an arbitrary node requires the masking of the input points at the current offset b_0 and the next step offset where:

$$b' = b_0 + \epsilon = \min_{\mathbf{P}_i}\{|p_i - 1|\} + 1, \tag{19}$$

where i is the queried axis. To determine the value of n, a summation is required over the two unmasked points sets. To accommodate for predicted offsets outside the current node's domain, in such an event, we zero the gradient stream and the loss gradient is solely determined by the dedicated penalty function of Eq. 10.

6 Applications and evaluation

We implemented our architecture in Tensorflow 2.9 and trained all our models using the Adam optimiser with learning rate 1.*e*-5 and the default settings. We also used batches of size 64, input point clouds of size 2048 and He-uniform weight initialisation for all layers excluding the last linear

Fig. 2 Illustration renderings of the scenes in the order of appearance in our experiments. Credits for scenes: Blue ©[3], Green ©[26], Red ©[19], Magenta ©[22]

one. All experiments were performed on an NVIDIA RTX 3080Ti with 12 GB of VRAM and an Intel i7 12700K CPU.

Our approach was evaluated on two discrete applications: nearest neighbour (Sect. 6.3) and ray tracing (Sect. 6.4). For all experiments we used the greedy tree inference (Sect. 4.3) due to its significantly faster hierarchy extraction at a similar query performance ($< 1\%$ difference).

6.1 Dataset

We trained and tested the model separately on a customly designed dataset consisting of 21 publicly available scenes that capture a variety of cases, from outdoor and indoor environments, to single-element meshes (Fig. 2). The primitive count in these scenes ranges from 57K to 18.5M. It is of no interest to apply the method to very small populations, since the build and query overheads exceed the benefits of applying a recursive evaluation. For the special case of the nearest neighbour query task, we choose 1 million area-weighted samples from the surfaces of each scene and use these as the initial dataset.

To train the network on point distributions that do not only correspond to entire environments, but also capture structure and primitive distributions at different detail levels, the training datasets are compiled as follows: for each scene in Fig. 2, from Contemporary Bathroom (a) to Modern Hall (q), we build a shallow k-d tree and sample 2048 points from every node, excluding the root. In the nearest neighbour case, we directly and uniformly draw 2048 samples from each node, while for the ray tracing case, we uniformly sample the triangle population, to retain the original primitive distribution and then, draw a single uniform sample from the selected primitive's surface. Finally, each point set is augmented with 2 random rotations. The training dataset consists of 3162 point clouds for the nearest neighbours case and 3114 for the ray tracing.

Even though this is an unsupervised method, we choose not to measure the quality directly on the training data, but

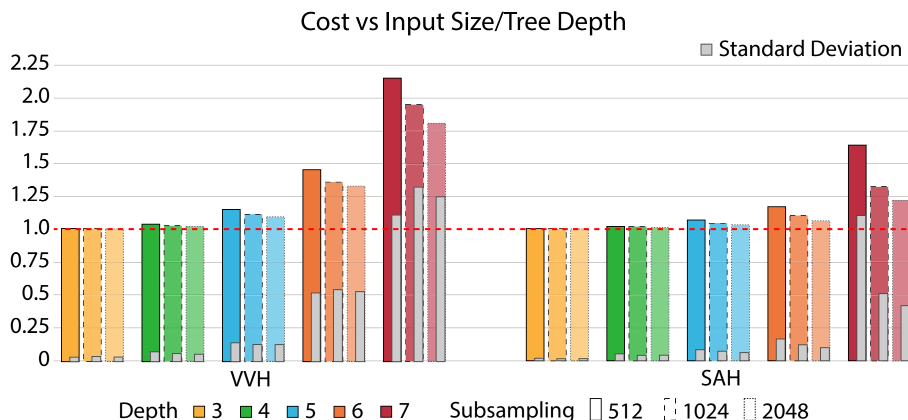

Fig. 3 Normalised VVH/SAH tree cost for a greedy k-d tree builder, using lower input sampling sizes, versus the entire data set (red line). Trees 3 to 7 levels deep are examined, with different sample counts (512, 1024 and 2048), averaged over all test scenes. Subsampled inputs for trees less than 5 and 6 levels deep, for the VVH and SAH cost functions, respectively, are able to properly represent the original point distributions and yield hierarchies with cost comparable to the baseline greedy k-d tree builder

rather construct a separate test set. The test set consists of 64 input point clouds sampled from the entire geometry of each scene (a) to (q), using the same sampling scheme as before, but with different scene parts from the ones used in the training set. Additionally, the test set includes 4 new scenes, from Breakfast Room (r) to Sports Car (u) (see Fig. 2). To conclude, validation is performed in 1344 point clouds for each task.

6.2 Impact of input size

To decide the appropriate depth of each inferred tree structure presented in the application sections in the main paper, we computed the average loss in quality, when constructing trees of increasing depth from a fraction of the input data. The results, averaged over all test scenes are shown in Fig. 3. In this experiment, to isolate the particular parameter, the (subsampled) input size, we constructed the trees with the greedy builder, instead of the neural one. We observe that for trees of depth up to 5 for the VVH cost (Fig. 3-left), and trees of depth up to 6 for the SAH cost (Fig. 3-right), most of the target quality is retained with input populations ranging from 1024 to 2048 samples, while the improvement of increasing further the input size is marginal.

Given the capabilities of our hardware setup, which limits the input size to 2048, it is not beneficial to train trees of 7 levels or more, as Fig. 3 confirms.

In order to train the model for deeper trees in each task, we hypothesise that not only the sample population should be increased, but also density priors on the input points should be given, something we plan to investigate in the future.

It is worth noting that for the given input size and tree depth, the inferred hierarchies are complete balanced trees.

6.3 Application: nearest neighbour search

Querying the nearest neighbours of points or gathering samples within a certain radius are common tasks in geometry processing [2] and global illumination algorithms, such as photon mapping [13] and its derivative methods. Such tasks typically involve constructing a k-d tree and optimising the VVH cost function (Eqs. 4 and 6). We set traversal and primitive intersection parameters to $c_t = 1.2$, $c_i = 1$, according to standard practice in the bibliography, and use a VVH radius of 10^{-4} (see $V(\mathbf{B})$ in Eq. 6). To balance between degradation in quality due to subsampling and tree depth, we train our model with 5 levels (2^4 leaves).

The typical approach to optimise such a function is to greedily evaluate the locally optimal alternative cost at each individual node. Since the VVH cost function is piece-wise constant, one needs to exhaustively evaluate partition splits only at the input point set member coordinates, for each splitting axis and keep the subdivision with the lowest cost. We use such a builder with a greedy cost evaluation as the baseline in our experiments. We use the inferred hierarchy of the neural builder as the topmost part of our hierarchy and fully populate the leaves and expand them with a greedy k-d tree builder. Then, we compare query performance against a corresponding k-d tree constructed entirely with the greedy builder.

Traversal. To benchmark the quality of our tree, we apply three query configurations as follows: (a) traversing the tree and recording the population in the resulting leaf, as this point-in-volume type of query is directly compatible with the VVH cost function, (b) k-nearest search, with $k = 1$, in order to avoid distorting the traversal time by overheads introduced by extra result container updates on the GPU side

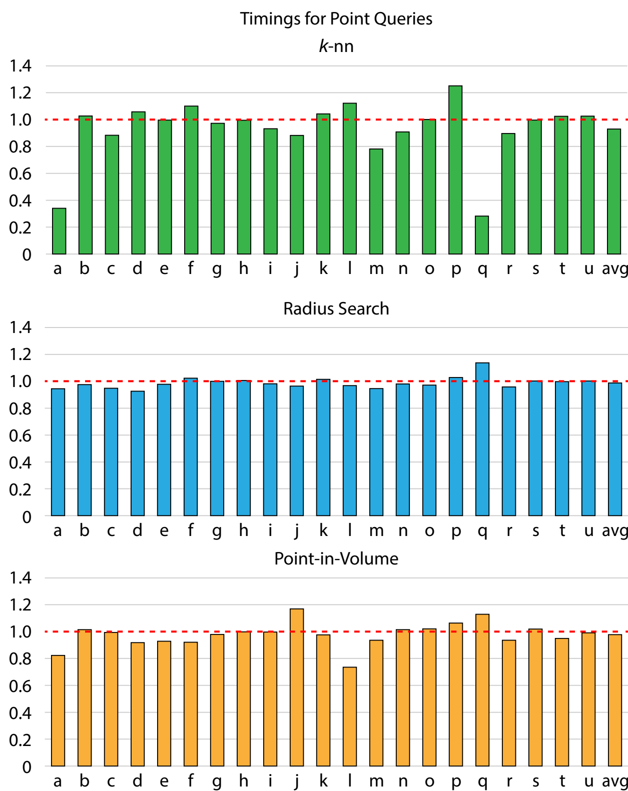

Fig. 4 Relative timings for point queries, where samples are uniformly distributed on the geometric surfaces. Results are normalised with respect to the reference greedy k-d tree implementation (red line)

and (c) radius search with $r = 1.e - 4$, by reporting the population in the queried domain.

For all query configurations we generate 10 batches of 1Mil. area weighted samples on the geometric surfaces of each scene. This distribution is characteristic for common applications, such as point cloud registration, particle tracing and spatial illumination caching. For both hierarchies we use the same node layout in memory and CUDA traversal kernel identical to the backtrack strategy employed by [25]. In Fig. 4 we report query times relative to the greedy standalone builder indicating that the final tree quality can be improved with our builder. Our results indicate that our approach, in most cases, performs better than the reference method, for all chosen tasks.

We also deem informative to mention that for the set of experiments that perform radius search queries, a builder using the VVH cost function of Eq. 6 expands a node's extents by a radius r. While the baseline k-d tree and the lower part of the hybrid hierarchy are computed using the specific search radius at hand, our network is trained once, with a constant value of $r = 1.e-4$. During the inference stage, using higher radii r than what the network has been trained with (here $r = \{0.5\%, 1\%, 2\%, 4\%\}$ of the scene's diagonal), has the negligible performance impact of $\pm 1\%$. This is attributed to the fact that at shallower tree depths, the relative node expansion due to different radii is not significant, affecting minimally the overall structure shape, in contrast to the significant extent changes that occur at the lower (and much smaller) hierarchy nodes.

Construction. To build the topmost part of the hierarchy, the neural builder requires a constant time of 2.5ms plus the negligible cost of 1ms to distribute the actual data points into the inferred leaves. Compared to a GPU k-d tree builder [36] for building the top part of the tree , using the same number of samples as the neural builder, either via exhaustive construction or binning (12 to 256 bins), our approach outperforms the GPU builder by $5.32\times$ and $3.33 \times -18.01\times$, respectively. From the remaining part of the tree, the expansion of each subtree hierarchy can be fully parallelised using the same GPU builder. In total, we obtain a speedup of $1.26\times$ on average to construct our hybrid tree over the GPU implementation with any building strategy.

6.4 Application: ray tracing

Ray tracing is nowadays used for offline and real-time rendering but also for other, non-image synthesis tasks, ranging from collision detection to geometry analysis. In this application category, ray-primitive acceleration data structures typically employ the *SAH* cost function (Eqs. 4 and 5), greedily (locally) evaluated at each node. For this task, we train our model for trees with 6 levels (2^5 leaves), for which the degradation in quality due to subsampling is marginal.

To build meaningful acceleration data structures for ray traversal, we first convert the 6-level k-d tree produced by the neural builder to a standard BVH layout. The conversion is done by exploiting the non-overlap-ping property of the k-d tree and assigning the corresponding bounds of clipped primitives to the associated neural leaves. This comes at a negligible cost of geometry duplication, which ranges from 0.4% to 7.6% for the Barcelona Pavilion (m) and Wooden Staircase (p) scenes, respectively. After the final hierarchy has been constructed and using the inferred connectivity, the bounding boxes of the upper tree are tightly re-fitted in a linear bottom-up pass. Next, we extend the 6-level tree with the publicly available implementation of ATRBVH [6] to obtain an end-to-end BVH hierarchy and we also do the same for the SBVH [29] implementation exposed through the Embree v3.13.4 [33] custom builder API.

We include performance comparisons for trees built entirely with both the SBVH and the ATRBVH algorithms. For both builders, we use the default settings and for the training/inference and the hierarchy construction stages, we use $c_t = 1.2$ and $c_i = 1$, as is typical in the bibliography. For every builder, we use the same node layout in memory and CUDA traversal kernel as discussed by [24], where we modify it to also exploit the non-overlapping bounds of the

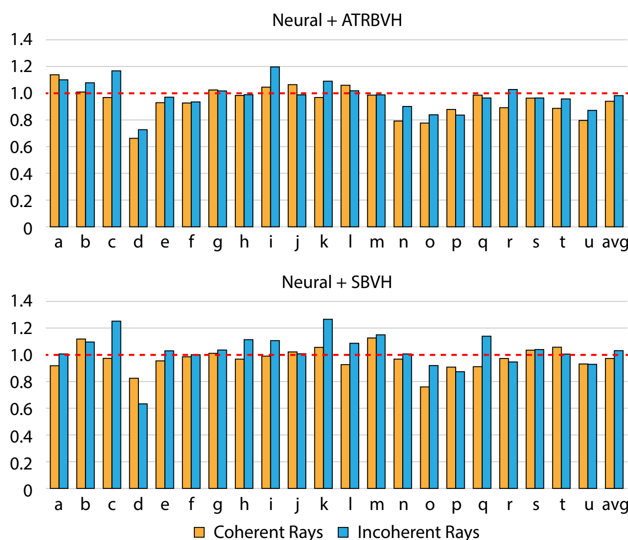

Fig. 5 Relative traversal timings of neural + ATRBVH (top) and neural + SBVH (bottom), separately for primary (yellow) and 4 diffuse indirect rays (blue). Times measured on path-traced scenes and normalised with respect to baseline ATRBVH and SBVH (red line), respectively

upper part of the tree to terminate rays early, when traversing it.

Traversal. To measure the performance of coherent and incoherent ray distributions in typical rendering applications, we employ path tracing using the viewpoints shown in Fig. 2, along with 3 additional meaningful viewports for each scene. We render the images at 3840 × 2160 resolution with 5 paths per-pixel—the primary rays followed by 4 purely diffuse indirect bounces—and record the average ray wavefront times over 5 independent iterations. In Fig. 5, we present query times relative to the standalone builders. Our results indicate that our builder is able to produce hierarchies of competitive or better performance, for both ray distributions.

Construction. Building the final hierarchy involves three main steps: first, the greedy inference for the top part of the tree (see Sect. 4.3), with a constant time of 5ms measured with the Tensorflow API in Python that encompasses overheads between tensor function calls. Here, constructing the top part of the tree with the GPU k-d tree builder (as in Sect. 6.3) using the same number of samples as the neural builder, we obtain 2.91× and 1.88 × −13.8× speedup for exhaustive construction and binning, respectively. Next, we assign the clipped bounds of the actual primitives to the associated leaves. This operation can be fully parallelised in the GPU and takes about 2ms on average for every scene, excluding Pavilion (m) and Power Plant (s) which require 5ms and 13ms, respectively. Finally, the lower hierarchy is constructed, using a standard builder, as explained next.

Expanding with the ATRBVH builder, we build all LBVH trees in a single kernel call and then optimise and collapse only once the expanded neural nodes. Since the ATRBVH builder works on the entire, partitioned input, the total build time increases, as expected, due to the segmented operations invoked in the kernels. The cumulative overhead of inferring the top part of the tree, distributing the primitives and invoking the builder increases the build time by a factor of 3× on average for all scenes. However, this large overhead is slightly misleading, as ATRBVH requires less than 8ms on average to build the final hierarchy for the relatively lighter scenes, and consequently, the neural inference roughly amounts to 55% of the total build time.

For the case of hierarchy expansion based on Embree's SBVH builder we exploit the CPU parallelism and employ its builder independently for every leaf. Similar to the previous case, there is an overhead from the input partitioning, added to the inference and the primitive distribution to leaves. For this case, the average total build time increases by a factor of 1.24× compared to the standalone builder. The smaller overhead compared to ATRBVH is attributed to the fact that the CPU standalone builder is slower. Additionally, the thread scheduling in the host side favours our method in several cases.

7 Discussion and conclusions

In this work, we proposed an unsupervised neural method to infer a spatial data structure through explicit optimisation of a recursive cost function. We demonstrated its expressiveness to generate low-cost solutions in two widely popular tasks and experimentally validated its consistency and generalisation capabilities. Despite the fact that our implementation for the data structure inference relies on a general-purpose API, the simplicity and size of the resulting network makes the neural builder an ideal candidate for a high-performance single-kernel GPU implementation.

Employing a hybrid builder to produce the top part of an acceleration data structure, our method can produce high-quality spatial splits, in short and constant time, regardless of the scene size, enabling early query termination and allowing for a fast third-party builder to efficiently operate by distributing the work on the sub-trees, even for large inputs. To handle prohibitively large inputs we applied uniform subsampling and demonstrated that, after embedding the original input, the hierarchy can still retain most of the original quality. To adequately handle local structure at arbitrary tree depths, either more elaborate input point priors or an adaptive input sampling scheme could be explored. Finally, an interesting future direction would be to investigate the application of recursively cascading neural-network-based trees to fully expand the tree since at its current state, naïvely sharing the same parameters for lower treelet neural blocks is expected to diminish the performance.

Acknowledgements This research was funded by the Hellenic Foundation for Research and Innovation (HFRI) under the "3rd Call for H.F.R.I. Research Projects to support Post-Doctoral Researchers" (Project No: 7310).

Funding Open access funding provided by HEAL-Link Greece.

Data availability Our code and datasets are publicly available at https://github.com/cgaueb/nss under the MIT license.

Declarations

Conflict of interest All authors declare that they have no conflicts of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
2. Besl, P.J., McKay, N.D.: A method for registration of 3-D shapes. IEEE Trans. Pattern Anal. Mach. Intell. **14**(2), 239–256 (1992)
3. Bitterli, B.: Rendering resources (2016)
4. Charles, R.Q., Su, H., Kaichun, M., Guibas, L.J.: Pointnet: Deep learning on point sets for 3d classification and segmentation. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 77–85 (2017)
5. Defferrard, M., Bresson, X., Vandergheynst, P.: Convolutional neural networks on graphs with fast localized spectral filtering. In: NIPS'16, pp. 3844–3852. Curran Associates Inc., Red Hook, NY, USA (2016)
6. Domingues, L.R., Pedrini, H.: Bounding volume hierarchy optimization through agglomerative treelet restructuring. In: Proceedings of the 7th Conference on High-Performance Graphics, HPG '15, pp. 13–20. Association for Computing Machinery, New York, NY, USA (2015)
7. Ganestam, P., Barringer, R., Doggett, M., Akenine-Möller, T.: Bonsai: rapid bounding volume hierarchy generation using mini trees. J. Comput. Graph. Tech. (JCGT) **4**(3), 23–42 (2015)
8. Goldsmith, J., Salmon, J.: Automatic creation of object hierarchies for ray tracing. IEEE Comput. Graph. Appl. **7**(5), 14–20 (1987)
9. Hanocka, R., Metzer, G., Giryes, R., Cohen-Or, D.: Point2mesh: a self-prior for deformable meshes. ACM Trans. Graph. **39**(4), 126:1–126:12 (2020)
10. Hendrich, J., Meister, D., Bittner, J.: Parallel BVH construction using progressive hierarchical refinement. Comput. Graph. Forum **36**(2), 487–494 (2017)
11. Huber, P.J.: Robust estimation of a location parameter. Ann. Math. Stat. **35**(1), 73–101 (1964)
12. Hunt, W., Mark, W.R., Fussell, D.: Fast and lazy build of acceleration structures from scene hierarchies. In: 2007 IEEE Symposium on Interactive Ray Tracing, pp. 47–54 (2007)
13. Jensen, H.W.: Global illumination using photon maps. In: Proceedings of the Eurographics Workshop on Rendering Techniques '96, pp. 21–30. Springer, Berlin (1996)
14. Kalojanov, J., Billeter, M., Slusallek, P.: Two-level grids for ray tracing on GPUs. Comput. Graph. Forum **30**(2), 307–314 (2011)
15. Karras, T., Aila, T.: Fast parallel construction of high-quality bounding volume hierarchies. In: Proceedings of the 5th High-Performance Graphics Conference, HPG '13, pp. 89–99. Association for Computing Machinery, New York, NY, USA (2013)
16. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings. OpenReview.net (2017)
17. Klokov, R., Lempitsky, V.S.: Escape from cells: deep kd-networks for the recognition of 3d point cloud models. In: 2017 IEEE International Conference on Computer Vision (ICCV), pp. 863–872 (2017)
18. Li, R., Li, X., Hui, K.H., Fu, C.W.: SP-GAN: sphere-guided 3d shape generation and manipulation. ACM Trans. Graph. **40**(4), 1–12 (2021)
19. Lumberyard, A.: Amazon lumberyard bistro, open research content archive (ORCA) (2017)
20. MacDonald, D.J., Booth, K.S.: Heuristics for ray tracing using space subdivision. Vis. Comput. **6**(3), 153–166 (1990)
21. Maturana, D., Scherer, S.: Voxnet: a 3d convolutional neural network for real-time object recognition. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 922–928 (2015)
22. McGuire, M.: Computer graphics archive (2017)
23. Meister, D., Bittner, J.: Parallel reinsertion for bounding volume hierarchy optimization. Comput. Graph. Forum **37**(2), 463–473 (2018)
24. Meister, D., Ogaki, S., Benthin, C., Doyle, M.J., Guthe, M., Bittner, J.: A survey on bounding volume hierarchies for ray tracing. Comput. Graph. Forum **40**(2), 683–712 (2021)
25. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISAPP International Conference on Computer Vision Theory and Applications, pp. 331–340. INSTICC Press (2009)
26. Pharr, M., Wenzel, J., Humphreys, G.: Scenes for pbrt-v3 (2016)
27. Pérard-Gayot, A., Kalojanov, J., Slusallek, P.: GPU ray tracing using irregular grids. Comput. Graph. Forum **36**(2), 477–486 (2017)
28. Riegler, G., Ulusoy, A.O., Geiger, A.: Octnet: learning deep 3d representations at high resolutions. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6620–6629 (2017)
29. Stich, M., Friedrich, H., Dietrich, A.: Spatial splits in bounding volume hierarchies. In: Proceedings of the Conference on High Performance Graphics 2009, HPG '09, pp. 7–13. ACM, New York, NY, USA (2009)
30. Wald, I.: On fast construction of sah-based bounding volume hierarchies. In: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, RT '07, pp. 33–40. IEEE Computer Society, USA (2007)
31. Wald, I., Günther, J., Slusallek, P.: Balancing considered harmful—faster photon mapping using the voxel volume heuristic—. Comput. Graph. Forum **23**(3), 595–603 (2004)
32. Wald, I., Havran, V.: On building fast kd-trees for ray tracing, and on doi2ng that in o(n log n). In: 2006 IEEE Symposium on Interactive Ray Tracing, pp. 61–69 (2006)

33. 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), 1–8 (2014)
34. Wang, P.S., Sun, C.Y., Liu, Y., Tong, X.: Adaptive O-CNN: a patch-based deep representation of 3d shapes. ACM Trans. Graph. **37**(6), 1–11 (2018)
35. Zaheer, M., Kottur, S., Ravanbhakhsh, S., Póczos, B., Salakhutdinov, R., Smola, A.J.: Deep sets. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS'17, pp. 3394–3404. Curran Associates Inc., Red Hook, NY, USA (2017)
36. Zhou, K., Hou, Q., Wang, R., Guo, B.: Real-time KD-tree construction on graphics hardware. ACM Trans. Graph. **27**(5), 1–11 (2008)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Mr. Iordanis Evangelou Iordanis Evangelou was born in Athens, Greece, in 1990. He received a 4-year B.Sc. in Computer Science from the Department of Informatics at the Athens University of Economics and Business and an M.Sc. in Computer Science from the same department. He is currently a doctoral student under the supervision of Georgios Papaioannou, and his main field of research is high-performance photo-realistic rendering and machine learning methods for computer graphics.

Prof. Georgios Papaioannou is an Associate Professor at the Department of Informatics of the Athens University of Economics and Business (AUEB) and head of the department's computer graphics group. His research is focused on real-time computer graphics algorithms, photorealistic rendering, shape analysis and geometry processing. Prof. Papaioannou has been the principal investigator for AUEB in many EU and nationally funded projects as well as R&D collaborations with the industrial sector. Prof. Papaioannou is also currently the director of the MSc programme in digital methods for the humanities and member of the UNESCO chair on digital methods for the humanities and social sciences. He has more than 85 publications in peer-reviewed international scientific journals, conference proceedings and volumes and is also a member of ACM SIGGRAPH and Eurographics associations.

Dr. Konstantinos Vardis is a postdoctoral fellow at the Department of Informatics of the Athens University of Economics and Business. His research work primarily focuses on real-time rendering techniques, global illumination and interactive ray tracing and has been published in peer-reviewed scientific journals and conference proceedings. Dr. Vardis has also participated in several national and European projects and has served as a reviewer and a program committee member in leading international conferences and journals. Furthermore, he has extensive industry experience, having delivered projects in a multitude of areas such as game development, large data volume visualisation, geoinformatics and GPU drivers.

Dr. Anastasios Gkaravelis was born in Larissa, Greece, in 1989. He received his 4-year B.Sc. in Computer Science from the Department of Informatics at the Athens University of Economics and Business and a Ph.D. in Computer Graphics in 2019 from the same department, under the supervision of Prof. Georgios Papaioannou. The subject of his doctoral thesis was "Efficient Algorithms for Inverse Lighting Design". His research interests are focused on computer graphics and, in particular, real-time and offline global illumination algorithms and photorealistic rendering.