Semestral Assignment 4: Improving kd-tree Quality for Ray Tracing at a Reasonable Cost

Jakub Profota

December 16, 2024

Abstract

This work implements a kd-tree spatial data structure with a cost model based on the surface area heuristic that uses binning sampling. The tree performs clipping on triangles straddling the splitting plane, generating new bounding boxes throughout the construction. Triangles whose surfaces do not overlap the bounding box of a node are pruned and discarded. Thus, the creation of empty leaf nodes is possible. The text starts with implementing a naive spatial median kd-tree and iteratively improves that until the final clipping pruning kd-tree is reached. All the trees created throughout the process are evaluated and compared regarding performance and statistics metrics.

Introduction

When rendering a scene with a method based on ray tracing, the essence of such algorithms is finding the nearest intersection of a ray with objects of the scene, if such an intersection exists. Although the definition of the query is trivial, there is a complex and challenging problem of finding the nearest intersections between thousands or even millions of triangles and thousands or even millions of rays, each corresponding to a single pixel of a commonly used frame buffer resolution, such as 800×800. The naive algorithm for ray tracing with $O(n)$ complexity computing intersections with all the triangles is not feasible even for offline methods. To restrict the number of intersection tests, a hierarchical spatial data structure is built to accelerate the queries. In this text, we focus on a kd-tree.

The kd-tree recursively partitions space with planes perpendicular to the axes of a coordinate system. Every inner node of the tree has a defined splitting plane position separating the space into two half-spaces, each included in one of its two children nodes. Splitting the space also partitions the triangles, assigning those overlapping the respective half-space to the respective child node. The leaf nodes contain a list of triangle references. The quality of the space splits can be improved by utilizing a cost model based on the surface area heuristic (SAH). Under some simplifying assumptions, minimizing this cost function enhances the quality of the data structure and improves query times. Since a high-quality kd-tree efficiently decomposes scene complexity and isolates empty space, the average intersection query has logarithmic complexity $O(\log N)$.

The construction time must also be considered for online, close to real-time ray tracing. SAH computation for finding the optimal cost is quite costly, so we look for an optimal trade-off between the kd-tree quality and the construction time. Instead of finding the best split, a sampling strategy such as binning is used. The space is partitioned together with the triangles into a set number of bins, each separated with a candidate splitting plane. SAH is then computed only for these candidates. Another technique to reduce the construction time is clipping, where bounding boxes of triangles straddling the splitting plane are split in two. This method improves SAH further down the recursion. Some newly created bounding boxes may not overlap with any half-spaces. Also, a leaf's half-space may overlap with the bounding boxes but not the triangle surfaces. In both of these cases, the triangles can be discarded

via pruning.

In the following text, we introduce and implement a naive spatial median split kd-tree, to which we iteratively add the aforementioned techniques. We start with a binning SAH strategy, followed by the clipping routine, and finish off with pruning. Finally, we test all the variants of the accelerated structure against each other and compare them.

Spatial median kd-tree

This hierarchical spatial data structure is built recursively in a top-down fashion. The construction method takes and stores the array of objects as its input. The recursive function then works only with the indices referencing the objects in the array, thus saving memory. Each object is initialized with its tight bounding box. Upon entering the tree construction method, the bounding box encompassing the entire space is created by unionizing all bounding boxes of objects. Space for arrays of nodes and object references is reserved, and an estimate of the triple the size of the object array is used. Using a preallocated array improves the performance of the kd-tree construction as allocations on the heap during the construction are avoided. The recursive function is called with all object references and the bounding box of the entire encompassing space.

Upon entering the recursive build function, a new node is created at the back of the array of nodes. Then, if the number of object references passed to the function is less than or the depth of the recursion is deeper than some threshold, a leaf node is created, and the recursion stops. If these criteria are not met, the build process finds a new splitting plane, splits the space of the current bounding box into two half-spaces, and partitions the object references accordingly. Finally, the recursion continues in the left and right child of the node.

When a leaf node is created, it is first marked as a leaf. Such information is stored in the two least significant bits of the index member of the node structure. Then, the total count of object references stored in this leaf node is appended to the kd-tree's array of object references. The leaf node's index points to this element of the array. All object references are then appended to the array. This way, there is no need to store the number of objects referenced directly in the node structure, which would lead to redundancy and unnecessary waste of memory since the internal node does not need this information.

The search for the splitting plane for our first kd-tree implementation is trivial, but we are going to improve on that further down the road. The position of the splitting plane is computed by simply taking the spatial median of the bounding box in the dimension of the largest extent. Since the splitting plane is always perpendicular to the coordinate system, we only need to store a single number, which represents the position of the plane along the chosen axis. This is stored as a floating point number in the node's data member. The axis of the splitting plane is stored in the two least significant bits of the index member at the same position as the leaf flag. In total, the node data structure occupies 8 bytes of memory, and it is shared by both internal and leaf nodes.

Upon choosing a new splitting plane, the space of the bounding box is split into two half-spaces by creating two new bounding boxes and setting the maximum of the left space and minimum of the right space to the value of the splitting plane position. Object references are then iterated, taking the bounding box of each of the referenced objects and checking if it overlaps the left and the right bounding box. If it does, such an object is sent further down the recursion. After this step, two arrays with object references intersecting the left and the right bounding boxes, respectively, are created. Evidently, if the bounding box of the object overlaps both of the bounding boxes, the reference to this object is stored in both of these arrays. Practically, only one additional array needs to be created since we can store object references of one of the half-spaces in place in the original object references array being passed to the recursive build function.

Finally, after partitioning the current node's space and sorting the object references, the recursive function is called for the left and right child of the node. We first call the recursion

for the left node, for which the corresponding entry in the array of nodes is created right away. This way, we do not need to store the index of the left child node since the index of that node is always the index plus one of the current node. After the recursion returns from the construction of the left node, we store the index of the right node that is going to be created right after at the back of the kd-tree's array of nodes. Then, the recursion for the right child commences.

The naive spatial median kd-tree is constructed, but to see if it works correctly, we need to implement a traversal algorithm.

Recursive traversal

The stack-based recursive traversal algorithm traverses the kd-tree by visiting all the necessary nodes exactly once. First, it checks whether the ray hits the encompassing bounding box of the spatial data structure. The traversal starts with the root node of the tree. The children of each visited node are classified as near and far child nodes based on their position relative to the splitting plane and the orientation of the intersecting ray. There are three possible cases for this classification. The ray intersects only the near child, or the far child, or the near child, and then the far child. In the third case, the stack stores the far child node if the traversal to the near child node does not find an intersection with any objects referenced throughout the traversal descent. The node is stored together with the entry and exit distance of the ray. The space requirements for the stack are minimal since the worst-case number of stored far child nodes is less than the depth of the kd-tree, which is, at maximum, the $\log_2$ of the number of objects but practically a little lower than that since we use a maximum depth threshold in the construction of the kd-tree.

There are three variants of the stack-based kd-tree traversal. Our implementation uses the variant to classify the near and far child nodes based on the ray direction. A thorough review of kd-tree traversal algorithms and pseudocode for our chosen algorithm is presented in [1]. The algorithm classifies the left node as near and the right node as far if the sign of the direction of the ray in the axis of the splitting plane is positive. For the negative sign, the nodes are swapped. The traversal of the nodes continues down the tree until a leaf node is reached. At that point, all the stored object references are iterated and checked for an intersection. If an intersection is found and it lies within the entry and exit distance of the ray in the node's bounding box, this intersection is valid. Furthermore, if the distance of the ray to the intersection is closer than some already found, this new intersection overrides the previous one. If one of the referenced objects of the leaf node is intersected, the algorithm exits the traversal and reports the closest intersection. In another case, the stack is popped for another node, and the same process applies.

Binning SAH strategy

The surface area heuristic finds the position of the splitting plane by minimizing a cost function that estimates the cost of ray traversal through the kd-tree under some simplifying assumptions. A good explanation of the motivation and derivation of SAH is in [2]. The SAH cost function is defined as the sum of three terms:

$$C_{SAH} = C_T + C_L + C_R$$

C_T represents the cost of traversing the interior node, and C_L and C_R are the costs for the left and right child, respectively. The costs of child nodes depend on the conditional geometric probability that a ray intersects the half-space of the child node while visiting the parent node. Such probability can be expressed as a ratio of the surface area of the child node's bounding box SA_{child} to the surface area of the parent's bounding box SA_{parent}:

$$p = \frac{SA_{child}}{SA_{parent}}$$

If the recursive build-up of the tree were bound to stop at these child nodes, then all the object references would need to be tested for an intersection in a kd-tree's traversal. The cost of the intersection test for such a node is then defined as a sum of computational time T_i for an intersection test of the i-th object over all

N object references:

$$C = \sum_{i=1}^{N} T_i$$

Thus, the costs of the left and right child nodes are:

$$C_L = p_L \cdot \sum_{j=1}^{N_L} T_j, \quad C_R = p_R \cdot \sum_{k=1}^{N_R} T_k$$

Substituting the costs and the probabilities in the SAH cost function with the aforementioned equations, we get the final formulae for the cost computation:

$$C_{SAH} = C_T + \frac{SA_L}{SA_{parent}} \cdot \sum_{j=1}^{N_L} T_j + \frac{SA_R}{SA_{parent}} \cdot \sum_{k=1}^{N_R} T_k$$

The goal is to build the kd-tree while minimizing the SAH cost recursively. Some terms in the equation can be omitted without changing the position of the minimum of the cost function. The C_T term is a constant factor that can be discarded. We can safely assume the computational time T_i is the same for all objects since all objects in our kd-tree are triangles and use the same intersection test method. We can then simplify the sum over N objects $\sum_{i=1}^{N} T_i$ by simply using the number N. Finally, the surface area of both the left and right child nodes is divided by the surface area of the parent SA_{parent}. Dividing both sides with the same number does not change the position of the minimum. We can omit that as well, obtaining the following relaxed form of the SAH cost function used in our implementation:

$$C_{SAH} = SA_L \cdot N_L + SA_R \cdot N_R$$

Even though we simplified the computation of SAH cost for a single splitting plane candidate, we would still need to compute the SAH cost for each candidate to find the optimal minimum. Such a computation could be very costly for many objects. While improving the final query times, the construction performance would suffer, potentially causing the kd-tree to be unusable for online rendering applications. However, we can sacrifice some precision in favor of the computation speed by sampling the space with a set number of samples and choosing the minimum only from these candidates. One of the techniques is called binning. Our implementation uses min-max binning, as described in [1].

The idea is to create two arrays with the size of the number of bins, one storing the count of starts of object bounding boxes and the second storing the number of ends. The start of a bounding box is defined as the box's minimum along the chosen axis, and the end is the maximum. Then, in one pass over all objects in a particular node, the corresponding start and end counts at the corresponding indices of the two arrays are incremented. After the arrays are filled with the numbers of starting and ending object bounding boxes, the second pass finds the best splitting plane position with the minimum cost. Before the sweep, the number of left object references is set to zero, and the number of right object references is set to the number of object references of the current node. These numbers are the N_L and N_R in the relaxed SAH cost function. When stepping to the next split position candidate, the count of starting bounding boxes in the corresponding bin index is added to the N_L, and the count of ending bounding boxes is subtracted from the N_R. This binning strategy can be applied either on the axis with the largest extent of the bounding box or on all three dimensions. The results section of this text compares both approaches.

Clipping

So far, the object reference with its bounding box overlapping both half-spaces has been passed to the recursive function call for both children. This approach does not account for cases where the bounding box overlaps the half-space, but the object's surface does not. The motivation for using a clipping technique is the reduced number of object references passed to the nodes in the recursion during the construction and the reduced number of object references stored in leaf nodes. There is a possibility of improved query times during the ray tracing and faster construction thanks to fewer references being passed around. Although the clipping routine is only beneficial

with pruning enabled, the pruning does not perform as well without the clipping. This section focuses on the clipping technique as described in [3], leaving the pruning for later.

The clipping is performed only on the triangles straddling the splitting plane, constructing two tight axis-aligned bounding boxes around the two parts of the cut triangle. The resulting boxes tightly bound the intersection of the triangle with the left and right half-space. Thus, the left bounding box is only passed to the recursive build function for the left child, and the right box is passed to the other. Each time the clipping routine clips a triangle, two new bounding boxes are created, and the original bounding box is no longer used during the construction. Since we use clipping only for triangles straddling the splitting plane, the algorithm is compact and performant.

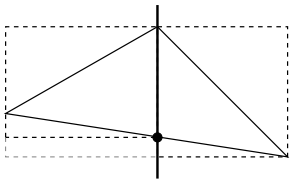
First, vertices are sorted along the split dimension. Then, one of the five possible cases is identified based on the relative position of the vertices in the half-spaces. These relative positions can be quickly computed by checking the sign of the vertex position V on the chosen axis d, subtracted by the position of the splitting plane P:

$$\text{sign}(V_d - P_d)$$

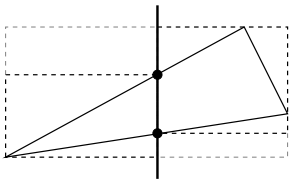
Then, if the vertex lies in the left half-space, the sign of such computation equals -1, and 1 for the right half-space. If the vertex lies directly on the splitting plane, then the sign of the relative position is 0.

The first two cases are relatively trivial; they happen if the triangle entirely lies in the left half-space and touches the right half-space with the right-most vertex or vice versa. In relative positions, these cases correspond to signs $(-1, -1, 0)$ and $(0, 1, 1)$, respectively. In such cases, the triangle is not clipped and is passed to the corresponding child recursion. Previously, the triangle touching the half-space with just one vertex would be sent to the corresponding child regardless.

The third case happens when the middle vertex lies on the splitting plane. Such a situation corresponds to relative position signs $(-1, 0, 1)$. An intersection of the splitting plane and the line segment defined by the vertex in the left half-space and the vertex in the right half-space is found, which will be used to create two new tight bounding boxes of the left and the right part of the triangle. Since the splitting plane is perpendicular to the coordinate system, finding such an intersection is trivial and reduces to a one-dimensional problem. The left and middle vertex and the plane intersection define the left bounding box. The right and middle vertex and the plane intersection define the right bounding box. The space of the original bounding box is not necessarily fully encapsulated in the combination of both the left and the right bounding box, as illustrated in the picture below.



The fourth and fifth cases happen if the left vertex lies in the left half-space and the middle and the right vertices lie in the right half-space and vice versa. These cases correspond to relative position signs $(-1, 1, 1)$ and $(-1, -1, 1)$. The algorithm determines the relative position of the middle vertex to obtain two line segments. The two segments connect the alone vertex in one half-space to the other two vertices in the other half-space. Then, two intersections of two line segments with the splitting plane are found trivially. Then, the two new tight bounding boxes are constructed. One contains the alone vertex and the two intersections, and the other contains the other two vertices and the two intersections.



The newly created bounding box is stored with the object's index in a global array of clipped triangles. The reference to be stored in a node is computed by adding the index of the clipped triangle in the global array to the total number of objects in the object list. If, during the construction or the kd-tree

traversal, the algorithm stumbles upon a triangle reference index larger than the number of objects, this index references a clipped triangle whose bounding box and index in the object list can be extracted from the global array easily by subtracting the size of the object list.

Pruning

Finally, the heavy lifting of the clipping routine can be adequately utilized by pruning the triangles whose bounding boxes do not overlap with any of the half-spaces. After creating the two tight bounding boxes, each is checked for overlap with its respective half-space. This technique reduces the number of object references passed around during construction and visited during the traversal.

We can even further improve the quality of the kd-tree by directly checking for an overlap of the triangle's surface. However, such a precise computation is significantly more costly than the fast overlap test of axis-aligned bounding boxes. Therefore, the surface overlap test is only utilized when creating a leaf node for each of the referenced objects using the algorithm based on the splitting axis theorem proposed in [4].

Configuration

The kd-tree runtime behavior can be configured through an environment file passed to the nanoGOLEM executable. The following snippet lists all new options added in the implementation:

```
RTCore {
  castShadowRays ON
}

KDTree {
  maxDepth 24
  maxLeafSize 4
  Binning {
    useBinning ON
    useDrivingAxis ON
    bins 16
  }
  useClipping ON
  usePruning ON
}
```

The `castShadowRays` option toggles casting shadow rays upon primary hit to compute shadows and light contributions. The `maxDepth` sets the maximum depth of the kd-tree at which the creation of a leaf node is forced regardless of the number of object references currently passed to the recursion. The `maxLeafSize` sets the threshold for creating a leaf node. A leaf node is created if the number of object references is less than this threshold. `useBining` option toggles between spatial median splitting and binning SAH strategy. If binning is enabled, `useDrivingAxis` dictates whether only the dimension of the largest extent of the bounding box is considered or whether binning searches in all three axes. The `bins` option sets the number of bins to use. `useClipping` toggles the clipping of triangles straddling the splitting plane, and `usePruning` toggles the pruning and discarding of triangles.

Results

The implementation is tested on 18 scenes. Ten of these are distributed by the professor of the subject and are mandatory. An additional seven models are downloaded from [5], particularly the well-known models frequently used in research papers and models with large numbers of triangles. The very last model is exported from the Blender modeling software. This model was used due to its low number of triangles and symmetry in debugging the kd-tree implementation.

Benchmarking is conducted on the NixOS distribution of GNU/Linux running on the 8th generation Intel Core i7-8565U notebook CPU. Although the CPU can reach clock speeds of 3.9 Gigahertz, the processor was, in reality, throttled down all the time due to the extensive benchmarking. The nanoGOLEM is compiled in release mode with `-O3` optimization using the GCC compiler, version 13.2.0. Each scene is rendered in 800×800 pixels at least 5 times. The reported results are the average of measured values. The ray tracer uses one primary ray per pixel and casts one shadow ray if the primary ray hits an object. The number of the primary ray hits and misses and the number of shadow ray hits and misses

for these primary rays that hit an object are shown in the table below. The rendered images can be found on the last pages of this text.

The `.view` scene files had to be altered or created, together with the code of the parser of this file, to add an ability to define a point light in the scene. Files defining a camera position and a point light for each scene are supplied together with the submitted codebase. These files end with `_L.view`, so they are not overwritten when the package of 10 mandatory models is downloaded and extracted to the scenes folder.

#	Scene	N_{tri}	P_{hit}	P_{miss}	S_{hit}	S_{miss}
1	A10	218,652	77,572	562,428	35,922	41,650
2	Armadillo	345,944	149,572	490,428	32,053	117,519
3	asianDragon	7,219,045	145,422	494,578	62,852	82,570
4	city2	75,420	271,822	368,178	166,960	104,862
5	city	68,497	377,970	262,030	202,140	175,830
6	conference	282,755	601,281	38,719	249,095	352,186
7	fforest	174,117	636,013	3,987	119,079	516,934
8	park	29,174	415,079	224,921	263,578	151,501
9	sibenik	80,479	639,989	11	237,258	402,731
10	teapots	200,748	557,241	82,759	152,189	405,052
11	buddha	1,087,474	132,282	507,718	59,835	72,447
12	hairball	2,880,000	385,890	254,110	369,636	16,254
13	lost_empire	242,870	449,011	190,989	161,709	287,302
14	powerplant	12,759,246	353,640	286,360	221,699	131,941
15	rungholt	6,704,264	574,363	65,637	359704	214659
16	san-miguel	9,980,699	640,000	0	391,277	248,723
17	sponza	262,267	625,811	14,189	221,290	404,521
18	suzanne	968	199,309	440,691	55,030	144,279

Table 1: 18 scenes used for benchmarking. The # column lists a number of the scene by which it is referenced in the following tables and figures. N_{tri} is the number of triangles in the scene. P_{hit} and P_{miss} are the number of primary ray hits and misses, respectively. Together, they sum up to the total number of primary rays cast, which is $800 \times 800 = 640,000$. S_{hit} and S_{miss} are the number of shadow ray hits and misses, respectively. Together, they sum up to the total number of shadow rays cast, which is the number of primary ray hits.

There are many combinations of the configuration options and, therefore, many ways to optimize the parameters for the best results. However, only nine different configurations are presented to optimize the time spent on this assignment. The results are presented in three sets of three tables, each containing three related kd-tree configurations. The first set focuses on the timings and the performance of the built data structure, the second set presents some quantitative statistics that are closely related to the performance of the tree, and the last set of tables shows statistics related to the depth and clipping and pruning capabilities of the kd-tree. The first table in each set lists three spatial median splitting kd-trees with different numbers of `maxDepth` and `maxLeafSize`. They do not clip or prune. The second table presents the results of various configurations of the SAH binning strategy. All these configurations share the same `maxDepth = 24` and `maxLeafSize = 8` settings. Again, these configurations do not clip or prune the triangles. Finally, the results of a binning kd-tree with `maxDepth = 24`, `maxLeafSize = 8`, `useBinning = ON`, `useDrivingAxes = ON`, and `bins = 8` are presented for only clipping, only pruning, and both clipping and pruning.

There are some visible artifacts in some scenes, as seen in the gallery of rendered images on the last pages of this text. Additionally, the implementation of the kd-tree boasts a high ratio of empty leaves, which could be collapsed through additional routines. Also, the clipping generates many new bounding boxes, many of which are further clipped and no longer referenced, consuming memory. Again, this could be worked upon further. However, due to the lack of time, the implementation is presented in its current state.

References

[1] M. Hapala and V. Havran, "Review: Kd-tree traversal algorithms for ray tracing," *Computer Graphics Forum*, vol. 30, no. 1, pp. 199–213, 2011.

[2] V. Havran and J. Bittner, "On improving kd-trees for ray shooting," 2002.

[3] A. Soupikov, M. Shevtsov, and A. Kapustin, "Improving kd-tree quality at a reasonable construction cost," pp. 67–72, Sep. 2008. DOI: 10.1109/RT.2008.4634623.

[4] T. Akenine-Möller, "Fast 3d triangle-box overlap testing," *J. Graph. Tools*, vol. 6, no. 1, pp. 29–33, Jan. 2002, ISSN: 1086-7651. DOI: 10.1080/10867651.2001.10487535.

[5] M. McGuire. "Computer graphics archive." (Jul. 2017), [Online]. Available: https://casual-effects.com/data.

T_B is the time spent building the kd-tree in seconds. T_R is the time spent rendering the scene in seconds. $perf$ is the performance in kilorays per second. N_{IT} is the average number of intersection tests per query. N_{TR} is the average number of internal nodes visited per query.

#	Spatial median 16/2					Spatial median 24/2					Spatial median 24/8				
	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}
1	0.045	0.84	762	96.14	4.86	0.12	0.23	2,782	9.1	7.08	0.087	0.22	2,909	11.94	5.73
2	0.13	1.5	427	117.13	6.93	0.237	0.36	1,778	11.54	9.97	0.215	0.37	1,730	14.69	9.22
3	1.293	20	32	2,404.63	7.42	1.876	1.1	581	81.8	10.2	1.881	1.1	581	82.57	10.09
4	0.016	0.39	1,641	30.14	5.54	0.164	0.31	2,065	10.8	9.66	0.051	0.29	2,207	16.72	6.57
5	0.015	0.82	780	82.16	15.24	0.066	0.47	1,362	23.05	19.47	0.037	0.47	1,362	33.54	15.18
6	0.049	3.1	206	365.40	25.5	0.148	1.3	492	86.81	36.62	0.112	1.2	533	94.84	31.32
7	0.036	6.7	96	756.78	26.96	0.297	1.3	492	75.33	39.74	0.07	1	640	76.88	27.39
8	0.007	1.8	356	187.61	23.3	0.081	0.88	727	51.85	42.33	0.022	0.98	653	93.55	27.25
9	0.017	9.4	68	1,016.55	25.09	0.061	1.3	492	81.25	44.89	0.035	1.3	492	93.3	40.76
10	0.036	3.7	173	459.76	29.59	0.068	0.73	877	47.99	33.6	0.056	0.67	955	62.66	22.48
11	0.219	2.2	291	186.8	4.72	0.45	0.44	1,455	17.74	8.18	0.417	0.44	1,455	20.32	7.57
12	0.747	9.8	65	963.19	21.62	2.718	3.1	206	189.2	46.39	2.657	3.3	194	202.45	43.36
13	0.047	1.3	492	121.59	14.76	0.188	0.59	1,085	15.89	22.25	0.104	0.64	1,000	36.52	17.96
14	2.423	77	8	1,689.12	19.18	3.759	3.1	206	287.46	29.58	3.729	3.2	200	291.71	28.04
15	1.104	17	38	1,801.55	19.2	2.208	1.3	492	56.4	32.47	2.26	1.5	427	67.48	31.52
16	1.963	-	-	-	-	2.701	13	49	1,202.06	63.32	2.816	14	46	1,217.92	56.22
17	0.051	11	58	1,137.15	31.94	0.232	1.5	427	82.08	50.81	0.138	1.5	427	107.01	34.71
18	0.002	0.27	2,370	15.5	9.27	0.073	0.57	1,123	27.53	15.78	0.002	0.25	2,560	21.64	5.54

Table 2: The depth of 24 leads to a dramatic decrease of object intersection tests while increasing internal nodes visits. This is desired as the internal traversal is cheaper than the intersection test.

#	Binning 8 (driving axis)					Binning 8					Binning 32				
	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}
1	0.221	0.2	3,200	10.54	4.58	0.338	0.16	4,000	7.72	4.28	0.416	0.15	4,267	7.59	4.11
2	0.386	0.38	1,684	14.47	8.38	0.749	0.36	1,778	13.46	7.96	0.921	0.37	1,730	12.9	7.79
3	4.072	0.68	941	26.66	9.72	8.265	0.77	831	27.32	9.61	8.613	0.62	1,032	24.81	9.52
4	0.087	0.29	2,207.0	16.38	5.81	0.039	0.22	2,909	13.1	5.49	0.043	0.21	3,048	12.86	5.15
5	0.085	0.43	1,488	29.5	12.77	0.115	0.35	1,829	22.85	11.42	0.152	0.32	2,000	19.61	11.24
6	0.346	1.1	582	87.6	26.0	0.482	0.89	719	67.42	22.64	0.531	0.81	790	60.95	18.94
7	0.157	0.86	744	59.82	22.64	0.269	0.73	877	47.18	22.28	0.373	0.67	955	38.5	19.53
8	0.049	0.82	780	74.45	22.52	0.062	0.61	1,049	51.92	20.58	0.079	0.57	1,123	48.82	19.99
9	0.083	1.0	640	68.95	33.44	0.125	0.86	744	50.79	30.97	0.164	0.76	842	43.21	28.0
10	0.146	0.69	928	61.98	20.44	0.253	0.61	1,049	53.97	18.78	0.315	0.6	1,067	52.34	18.51
11	0.883	0.41	1,561	15.46	6.79	1.713	0.39	1,641	14.31	6.49	1.925	0.38	1,684	13.55	6.28
12	9.555	3.3	194	154.26	37.3	15.417	2.9	221	133.98	36.53	12.725	2.2	291	110.38	35.07
13	0.216	0.59	1,085	29.42	15.63	0.37	0.5	1,280	23.71	15.09	0.454	0.47	1362	22.03	13.69
14	8.646	1.6	400	116.35	23.18	17.22	1.2	533	65.98	21.66	15.04	0.84	762	48.76	20.03
15	4.335	0.48	1,333	46.4	28.33	8.286	1.1	582	39.95	26.25	9.008	0.91	703	32.54	24.52
16	6.024	3.7	173	233.81	54.65	11.304	3.3	194	204.46	50.79	11.515	2.8	229	163.47	48.59
17	0.272	1.2	533	75.57	33.15	0.423	1.1	582	67.83	29.33	0.538	1.1	582	71.95	26.97
18	0.002	0.21	3,048	16.79	5.33	0.002	0.2	3,200	15.62	5.36	0.003	0.2	3,200	14.91	4.88

Table 3: Using the driving axis is beneficial for the build time while maintaining high ray-tracing performance. Increasing the number of bins has no negative effect on the build times.

#	Clipping					Pruning					Clipping pruning				
	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}	T_B	T_R	$perf$	N_{IT}	N_{TR}
1	0.423	0.15	4,267	6.42	4.04	0.518	0.13	4,923	5.61	4.11	0.473	0.15	4,267	6.01	4.02
2	0.99	0.37	1,730	11.94	7.66	0.943	0.32	2,000	9.8	7.8	1.098	0.36	1,778	10.62	7.64
3	10.23	0.65	985	22.8	9.48	9.834	0.56	1,143	21.13	9.52	11.276	0.63	1,016	21.92	9.45
4	0.048	0.22	2,909	12.83	5.15	0.048	0.21	3,048	12.86	5.15	0.055	0.23	2,783	12.83	5.15
5	0.17	0.33	1,939	16.42	10.76	0.203	0.31	2,065	15.46	11.24	0.188	0.32	2,000	15.68	10.74
6	0.533	0.65	985	39.06	17.32	0.684	0.63	1,016	40.4	18.94	0.587	0.63	1,016	37.09	17.19
7	0.368	0.58	1,103	30.39	19.22	0.436	0.56	1,143	29.71	19.54	0.432	0.64	1,000	29.04	19.31
8	0.078	0.55	1,164	41.74	19.49	0.101	0.55	1,164	38.23	20.01	0.089	0.54	1,185	40.77	19.47
9	0.187	0.74	865	38.15	27.27	0.213	0.71	901	35.24	28.03	0.213	0.72	889	37.2	27.2
10	0.386	0.64	1,000	47.23	20.21	0.369	0.62	1,032	44.5	20.48	0.427	0.65	985	45.53	20.19
11	2.314	0.39	1,641	11.44	6.16	2.351	0.33	1,939	9.86	6.29	2.688	0.38	1,684	10.33	6.13
12	11.301	1.9	337	75.07	32.87	16.996	1.9	337	71.91	35.08	12.723	1.8	356	70.53	32.62
13	0.559	0.48	1,333	21.32	13.7	0.601	0.47	1,362	21.13	13.69	0.618	0.47	1,362	21.24	13.69
14	16.77	0.86	744	34.32	19.27	16.978	0.72	889	31.99	20.03	17.765	0.74	865	33.58	19.2
15	12.068	1.1	582	30.75	24.53	10.741	0.96	667	30.69	24.53	13.674	1.0	640	30.38	24.54
16	13.503	2.7	237	139.93	45.85	14.279	2.5	256	132.94	48.67	14.56	2.5	256	135.87	45.68
17	0.606	1.1	582	58.25	26.54	0.658	0.93	688	52.0	26.99	0.655	1.0	640	56.51	26.38
18	0.002	0.17	3,765	11.71	4.68	0.004	0.16	4,000	10.97	4.88	0.003	0.17	3,765	11.11	4.67

Table 4: Unfortunately, the benchmarking shows that the additional computation cost of both clipping and pruning does not produce a tree of better quality than simply performing only pruning.

N_I is the number of internal nodes in the kd-tree. N_L is the number of leaves. N_{TP} is the total number of object primitive references in the leaves. N_{AP} is the average number of object primitive references in one leaf. *mem* is the approximate memory consumption in megabytes.

#	Spatial median 16/2					Spatial median 24/2					Spatial median 24/8				
	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*
1	8,226	8,227	318,614	39	3.031	453,964	453,965	2,178,531	4.8	18.382	130,716	130,717	1,240,304	9.5	8.838
2	8,841	8,842	482,108	55	4.635	410,736	410,737	1,773,601	4.3	16.827	263,857	263,858	1,618,413	6.1	13.703
3	6,824	6,825	7,751,662	1,100	84.768	318,757	318,758	11,073,003	35	102.945	301,836	301,837	11,071,719	37	102.679
4	32,208	32,209	300,312	9.3	2.302	1,439,952	1,439,953	4,754,128	3.3	44.899	238,443	238,444	1,438,012	6	10.49
5	10,745	10,746	153,026	14	1.3	364,836	364,837	1,503,618	4.1	12.913	91,149	91,150	824,197	9.0	5.346
6	11,054	11,055	444,199	40	4.053	535,020	535,021	3,439,103	6.4	25.054	171,256	171,257	2,340,437	14	14.246
7	20,786	20,787	316,708	15	2.919	2,191,686	2,191,687	12,238,838	5.6	89.606	121,384	121,385	1,261,042	10	8.425
8	15,377	15,378	101,839	6.6	0.892	692,166	692,167	2,109,924	3	20.963	65,485	65,486	460,945	7.0	3.186
9	9,372	9,373	140,402	15	1.321	333,161	333,162	1,176,819	3.5	11.128	57,838	57,839	464,670	8.0	3.445
10	4,860	4,861	248,288	51	2.567	180,962	180,963	846,805	4.7	8.03	44,818	44,819	532,093	12.0	4.38
11	16,545	16,546	1,416,871	86	13.995	827,071	827,072	4,670,103	5.6	41.076	557,197	557,198	4,266,877	7.7	34.842
12	35,499	35,500	6,732,292	190	48.318	4,945,941	4,945,942	71,488,275	14	385.441	4,101,671	4,101,672	69,862,637	17	364.846
13	21,724	21,725	452,395	21	3.965	1,039,761	1,039,762	3,134,005	3	32.304	266,425	266,426	1,716,114	6.4	13.344
14	3,983	3,984	14,186,760	3,600	151.535	325,942	325,943	25,854,789	79	201.958	238,091	238,092	25,546,826	110	199.176
15	27,335	27,336	7,773,560	280	81.302	2,147,773	2,147,774	21,137,850	9.8	170.393	1,908,167	1,908,168	21,093,404	11	166.526
16	-	-	-	-	-	946,753	946,754	17,164,950	18	158.702	423,653	423,654	15,895,536	38	144.524
17	25,833	25,834	490,081	19	4.339	1,245,099	1,245,100	5,736,844	4.6	46.964	332,240	332,241	2,905,342	8.7	19.321
18	11,717	11,718	39,008	3.3	0.376	759,513	759,514	2,394,896	3.2	23.576	5,833	5,834	40,934	7	0.275

Table 5: Increasing the depth of the tree dramatically decreases the average number of object primitive references in one leaf. This directly translates to the decreased number of object intersection tests as seen on the previous page.

#	Binning 8 (driving axis)					Binning 8					Binning 32				
	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*
1	295,076	295,077	2,596,005	8.8	17.1	199,184	199,185	1,659,752	8.3	11.72	223,830	223,831	1,826,273	8.2	12.825
2	321,599	321,600	1,784,421	5.5	15.446	310,977	310,978	1,715,372	5.5	14.981	318,877	318,878	1,748,485	5.5	15.259
3	1,537,949	1,537,950	17,864,230	12	151.686	1,421,690	1,421,691	17,259,274	12	147.246	1,566,865	1,566,866	17,559,120	11	151.017
4	204,079	204,080	1,215,228	6	9.011	21,555	21,556	95,514	4.4	1.332	19,234	19,235	92,472	4.8	1.279
5	147,204	147,205	1,130,492	7.7	7.559	87,081	87,082	585,493	6.7	4.355	101,205	101,206	674,658	6.7	4.958
6	390,310	390,311	5,691,710	15	31.162	216,894	216,895	3,326,664	15	18.894	215,198	215,199	3,077,771	14	17.924
7	190,826	190,827	1,620,383	8.5	11.089	163,725	163,726	1,363,195	8.3	9.592	202,038	202,039	1,661,524	8.2	11.451
8	100,141	100,142	712,929	7.1	4.797	56,992	56,993	372,208	6.5	2.693	57,624	57,625	379,334	6.6	2.734
9	132,902	132,903	897,068	6.7	6.486	87,149	87,150	549,523	6.3	4.31	103,302	103,303	654,430	6.3	5.019
10	165,679	165,680	1,149,108	6.9	8.99	142,588	142,589	962,908	6.8	7.843	158,247	158,248	1,029,024	6.5	8.39
11	933,294	933,295	5,631,579	6	47.187	889,544	889,545	5,357,906	6	45.324	901,386	901,387	5,389,726	6	45.678
12	5,049,382	5,049,383	162,446,831	32	735.753	4,644,080	4,644,081	162,844,504	35	729.337	4,442,456	4,442,457	90,170,457	20	447.6
13	351,940	351,941	2,428,341	6.9	17.682	295,523	295,524	1,763,150	6	14.053	297,915	297,916	1,806,319	6.1	14.272
14	1,075,825	1,075,826	46,606,274	43	295.006	802,906	802,907	28,462,364	35.0	220.624	1,086,990	1,086,991	29,121,592	27	228.302
15	2,865,272	2,865,273	27,757,418	9.7	209.856	2,789,489	2,789,490	26,154,628	9.4	201.831	2,812,547	2,812,548	27,642,106	9.8	208.474
16	1,279,392	1,279,393	26,195,556	20	199.727	1,090,306	1,090,307	22,762,546	21	183.114	1,380,006	1,380,007	24,800,473	18	196.319
17	386,710	386,711	2,818,401	7.3	19.979	291,693	291,694	1,997,145	6.8	15.048	299,945	299,946	2,070,772	6.9	15.501
18	3,305	3,306	20,025	6.1	0.146	2,863	2,864	16,701	5.8	0.125	2,834	2,835	17,836	6.3	0.129

Table 6: Major advantage of considering all three axes is the reduced number of leaf nodes and reduced memory consumption of the data structure.

#	Clipping					Pruning					Clipping pruning				
	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*	N_I	N_L	N_{TP}	N_{AP}	*mem*
1	140,714	140,715	884,982	6.3	47.649	223,830	223,831	958,891	4.3	9.371	134,601	134,602	809,844	6	42.616
2	285,304	285,305	1,411,996	4.9	73.876	318,877	318,878	1,470,061	4.6	14.099	280,825	280,826	1,321,050	4.7	68.873
3	1,503,257	1,503,258	16,203,473	11	647.968	1,566,865	1,566,866	16,111,430	10	144.856	1,490,336	1,490,337	15,735,674	11	620.904
4	18,974	18,975	89,495	4.7	3.384	19,234	19,235	91,132	4.7	1.274	18,909	18,910	89,232	4.7	3.359
5	68,326	68,327	369,029	5.4	20.989	101,205	101,206	396,099	3.9	3.854	65,960	65,961	337,127	5.1	18.883
6	132,346	132,347	1,180,305	8.9	66.773	215,198	215,199	1,191,898	5.5	10.513	125,121	125,122	1,074,135	8.6	57.706
7	121,508	121,509	774,764	6.4	42.537	202,038	202,039	827,857	4.1	8.076	116,803	116,804	701,165	6	37.966
8	33,805	33,806	181,664	5.4	10.576	57,624	57,625	199,262	3.5	2.013	32,047	32,048	159,487	5	9.326
9	76,952	76,953	418,347	5.4	23.45	103,302	103,303	444,860	4.3	4.193	74,537	74,538	387,545	5.2	21.417
10	139,219	139,220	811,961	5.8	42.32	158,247	158,248	826,862	5.2	7.567	136,804	136,805	757,578	5.5	39.17
11	787,507	787,508	4,243,789	5.4	219.551	901,386	901,387	4,442,581	4.9	41.752	767,641	767,642	3,966,364	5.2	203.525
12	2,988,699	2,988,700	26,125,760	8.7	1,866.848	4,442,456	4,442,457	24,974,277	5.6	194.408	2,792,887	2,792,888	21,796,969	7.8	1,477.234
13	239,810	239,811	1,291,031	5.4	74.838	297,915	297,916	1,398,988	4.7	12.718	227,772	227,773	1,231,018	5.4	69.093
14	899,092	899,093	24,617,909	27	883.131	1,086,990	1,086,991	24,676,259	23	210.829	882,895	882,896	24,214,096	27	847.642
15	2,784,206	2,784,207	24,024,664	8.6	1,244.728	2,812,547	2,812,548	23,646,191	8.4	193.231	2,764,899	2,764,900	23,134,277	8.4	1,172.446
16	1,224,684	1,224,685	21,138,487	17	834.554	1,380,006	1,380,007	20,838,198	15	180.817	1,199,561	1,199,562	20,407,452	17	780.979
17	217,336	217,337	1,249,824	5.8	69.58	299,945	299,946	1,341,342	4.5	12.604	209,431	209,432	1,149,784	5.5	62.813
18	1,089	1,090	5,529	5.1	0.315	2,834	2,835	8,028	2.8	0.089	1,046	1,047	4,838	4.6	0.28

Table 7: Due to the clipping routine generating two new bounding boxes for each triangle straddling the splitting plane, the memory consumption is significantly higher. Since no routine discarding unused bounding boxes is implemented, the memory is full of useless data which fills the cache and potentially slows down the performance of the data structure.

R_E is the ratio of empty leaves to the total number of leaves. D_{max} is the maximum depth of the tree. D_{avg} is the average depth of leaf nodes. N_{CL} is the number of clipped triangles. N_{PR} is the number of pruned triangles.

#	Spatial median 16/2					Spatial median 24/2					Spatial median 24/8				
	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}
1	0.32	16	16	0	0	0.15	24	23	0	0	0.11	24	22	0	0
2	0.36	16	15	0	0	0.26	24	23	0	0	0.14	24	23	0	0
3	0.36	16	15	0	0	0.37	24	23	0	0	0.34	24	23	0	0
4	0.27	16	16	0	0	0.23	24	23	0	0	0.13	24	21	0	0
5	0.28	16	15	0	0	0.22	24	23	0	0	0.17	24	22	0	0
6	0.24	16	15	0	0	0.21	24	23	0	0	0.16	24	23	0	0
7	0.18	16	16	0	0	0.026	24	24	0	0	0.062	24	23	0	0
8	0.21	16	15	0	0	0.19	24	23	0	0	0.18	24	22	0	0
9	0.22	16	15	0	0	0.26	24	23	0	0	0.2	24	23	0	0
10	0.21	16	15	0	0	0.27	24	23	0	0	0.21	24	23	0	0
11	0.35	16	16	0	0	0.26	24	23	0	0	0.17	24	23	0	0
12	0.1	16	16	0	0	0.19	24	24	0	0	0.12	24	24	0	0
13	0.34	16	16	0	0	0.34	24	23	0	0	0.13	24	22	0	0
14	0.32	16	15	0	0	0.19	24	24	0	0	0.18	24	24	0	0
15	0.22	16	16	0	0	0.29	24	24	0	0	0.2	24	24	0	0
16	-	-	-	-	-	0.27	24	23	0	0	0.21	24	23	0	0
17	0.24	16	16	0	0	0.14	24	23	0	0	0.079	24	23	0	0
18	0.082	16	15	0	0	0.019	24	24	0	0	0.012	24	19	0	0

Table 8: The ratio of empty leaves is lower for the spatial median split tree with the `maxLeafSize = 8`. In many scenes, most of the leaves do not reach the maximum depth of the tree.

#	Binning 8 (driving axis)					Binning 8					Binning 32				
	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}
1	0.089	24	23	0	0	0.1	24	22	0	0	0.093	24	22	0	0
2	0.11	24	22	0	0	0.11	24	22	0	0	0.11	24	22	0	0
3	0.15	24	23	0	0	0.14	24	23	0	0	0.16	24	23	0	0
4	0.12	24	21	0	0	0.23	24	17	0	0	0.21	24	17	0	0
5	0.15	24	22	0	0	0.19	24	22	0	0	0.18	24	22	0	0
6	0.1	24	23	0	0	0.11	24	23	0	0	0.096	24	23	0	0
7	0.083	24	22	0	0	0.096	24	22	0	0	0.089	24	23	0	0
8	0.14	24	22	0	0	0.17	24	21	0	0	0.16	24	21	0	0
9	0.17	24	22	0	0	0.19	24	22	0	0	0.16	24	22	0	0
10	0.13	24	23	0	0	0.15	24	22	0	0	0.14	24	22	0	0
11	0.11	24	23	0	0	0.11	24	23	0	0	0.11	24	23	0	0
12	0.11	24	24	0	0	0.14	24	24	0	0	0.18	24	24	0	0
13	0.11	24	22	0	0	0.14	24	21	0	0	0.14	24	21	0	0
14	0.16	24	24	0	0	0.2	24	23	0	0	0.21	24	23	0	0
15	0.17	24	23	0	0	0.22	24	23	0	0	0.16	24	23	0	0
16	0.15	24	23	0	0	0.16	24	23	0	0	0.14	24	23	0	0
17	0.1	24	23	0	0	0.12	24	22	0	0	0.11	24	22	0	0
18	0.045	24	18	0	0	0.055	24	18	0	0	0.028	24	18	0	0

Table 9: There is not much difference between the three binning configuration variants.

#	Clipping					Pruning					Clipping pruning				
	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}	R_E	D_{max}	D_{avg}	N_{CL}	N_{PR}
1	0.15	24	22	749,051	0	0.26	24	22	0	867,382	0.19	24	22	729,301	137,257
2	0.18	24	21	1,134,706	0	0.19	24	22	0	278,424	0.22	24	21	1,121,345	146,239
3	0.21	24	23	9,429,580	0	0.26	24	23	0	1,447,690	0.26	24	23	9,390,685	874,055
4	0.22	24	16	39,704	0	0.21	24	17	0	1,340	0.22	24	16	39,534	497
5	0.2	24	21	333,426	0	0.29	24	22	0	278,559	0.23	24	21	325,094	56,415
6	0.15	24	22	1,079,740	0	0.36	24	23	0	1,885,873	0.18	24	22	1,043,695	252,189
7	0.18	24	22	674,402	0	0.34	24	23	0	833,667	0.22	24	22	659,676	132,627
8	0.18	24	20	169,233	0	0.31	24	21	0	180,072	0.22	24	20	163,808	33,493
9	0.17	24	22	371,163	0	0.23	24	22	0	209,570	0.19	24	22	364,716	57,476
10	0.19	24	22	657,887	0	0.23	24	22	0	202,162	0.23	24	22	649,373	92,507
11	0.16	24	23	3,380,526	0	0.2	24	23	0	947,145	0.21	24	22	3,329,844	450,954
12	0.26	24	23	31,666,183	0	0.45	24	24	0	65,196,180	0.31	24	23	30,505,886	11,588,911
13	0.12	24	21	1,190,880	0	0.14	24	21	0	407,331	0.12	24	21	1,164,907	146,325
14	0.25	24	23	12,650,336	0	0.33	24	23	0	4,445,333	0.26	24	23	12,582,561	1,127,414
15	0.16	24	23	19,670,208	0	0.16	24	23	0	3,995,915	0.16	24	23	19,594,072	2,413,540
16	0.16	24	23	12,267,848	0	0.22	24	23	0	3,962,275	0.18	24	23	12,185,096	1,758,076
17	0.14	24	22	1,100,733	0	0.21	24	22	0	729,430	0.17	24	22	1,076,633	189,114
18	0.083	24	14	4,983	0	0.26	24	18	0	9,808	0.14	24	14	4,849	979

Table 10: The configurations with clipping enabled show the extraodinary amount of clipped triangles, in some scenes being even a multiple of the total number of triangles. It is interesting to see that the number of pruned triangles is significantly lower than when also the clipping routine is applied. This could be to a fact that a triangle is pruned much sooner in the recursion, thus preventing all the children from pruning the same single triangle later on.

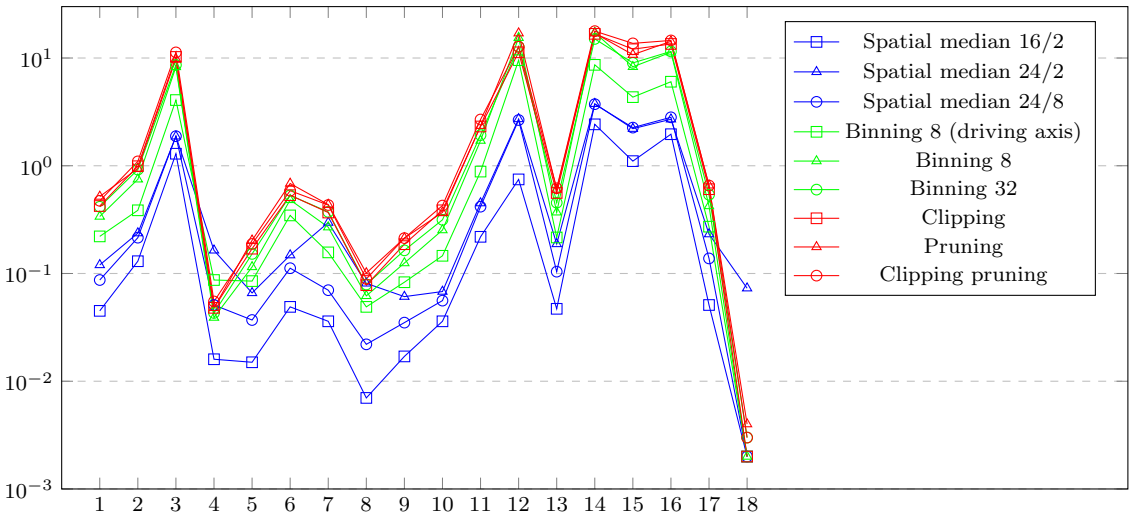


Figure 1: Build times for 9 tested kd-tree configurations in seconds. The x-axis represents the scene number, the y-axis the build time in seconds in logarithmic scale. In some of the scenes, more complex binning, clipping and pruning configurations surpass the simple spatial median split tree. The binning configuration using the driving axis is faster than the other two binning configurations considering all the axes. Combinations of clipping and pruning take a little more time than just binning, but the difference is not significant. The clipping pruning combinations also basically do not differ in the build time.

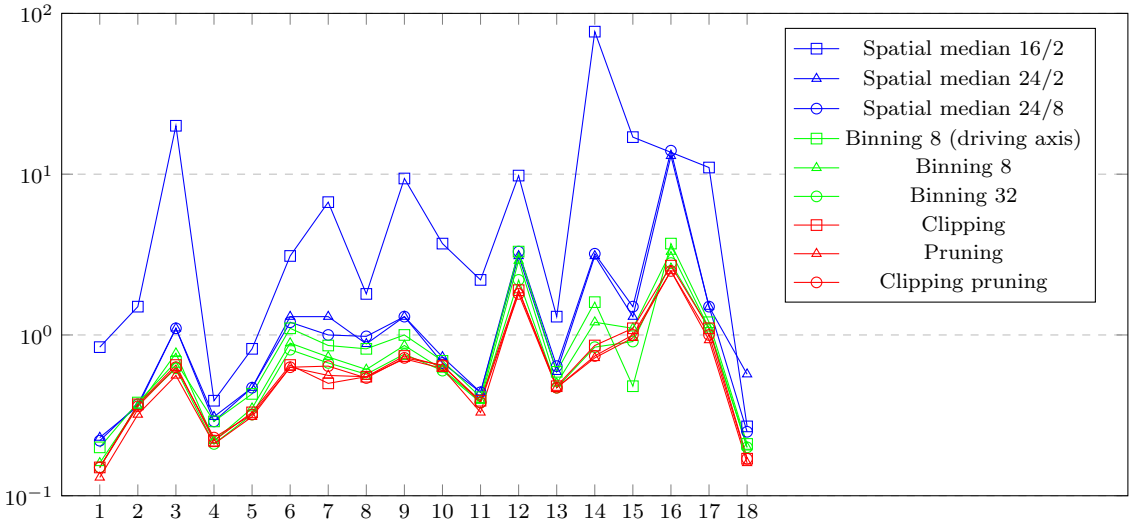


Figure 2: The clipping pruning configurations are the most performant in terms of the ray-tracing time. They surpass binning configurations while keeping the build time at a similar level.