Computer Graphics International 2011

The rkd-Tree: An Improved kd-Tree for Fast n-Closest Point Queries in Large Point Sets

Robert F. Tobler

Abstract The kd-tree is used in various applications, such as photon simulation with photon maps, or normal estimation in point sets for reconstruction, in order to perform fast n-closest neighbour searches in huge, static data sets of point sets of arbitrary dimensions. In a number of cases, where lower dimensional point sets are embedded in higher dimensional spaces, it has been shown that the vantage point tree (vp-tree) can significantly outperform the kd-tree. In this paper we introduce the rkd-tree, a modified version of the kd-tree that applies ideas from the vp-tree to the kd-tree. This improved kd-tree version is shown to outperform both the kd-tree and the vp-tree in a number of artificial and real test-cases.

Keywords nearest neighbour search · vp-tree · kd-tree

1 Introduction

Searching the closest point to a query point within a large point set is a typical task in a number of computer graphics applications. Here are two examples:

In photon maps that store all simulated photons in a forward tracing light simulation, the photons are typically stored in a searchable data structure so that the subsequent illumination estimation can quickly estimate the illumination of a query point. This is done by performing a search for the n closest points to the query point, and thereby estimating the photon density around the query point.

In reconstruction of 3D scenes based on laser scans or photos, the raw data or an intermediate data set consists of a large number of points on the surfaces of the objects that need to be reconstructed. In order to compute the surface normals and reconstruct surfaces, it is necessary to quickly find the n-closest points to each of the points in the data set.

In both of the cases the number of points in the data set to query is very large (often more than 10^6 points), and the number of queries that have to be performed is also large. Oftentimes a kd-tree that recursively partitions space along the dimensions of the search space is used to store the points and allow a fast algorithm for n-closest point queries. A highly optimized version has been published with the introduction of photon maps by Jensen [1].

In 1990 Yianilos introduced the vantage point tree (vp-tree) [2] that use a different strategy based on so-called vantage points, to recursively partition the search space. It has been shown that there are search application where the vp-tree significantly outperforms the kd-tree [3].

In this paper we will introduce an improvement to the kd-tree that is based on the vp-tree that we call the rkd-tree, and we will show a number of search scenarios where the rkd-tree outperforms both the kd-trees and the vp-tree.

2 State of the art

2.1 The kd-tree

The kd-tree has been introduced by Friedman and Bentley [4], [5], [6], and is often used for searches in low-dimensional spaces. In order to build a kd-Tree, the database of points is recursively bisected in order to build a binary search tree. Each bisection is performed

Robert F. Tobler
VRVis Research Center, Vienna, Austria
E-mail: rft@vrvis.at

at the median of all data points in a selected dimension. The selection of which dimension to bisect can be either be fixed (e.g. by rotating through all dimensions), or optimized by taking the dimension with the larges spread in the data. In figure 1 the partitioning of an example two dimensional point set is shown.

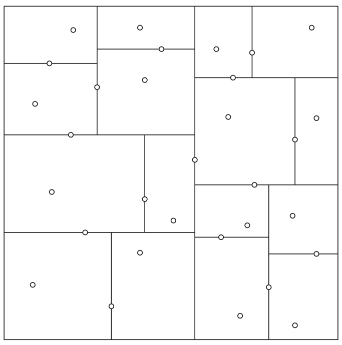

Fig. 1 Two dimensional example kd-tree decomposition.

Although a kd-tree can be represented in memory in the typical manner of a binary search tree by using nodes that contain references to their two sub-nodes, the observation that all bisections are performed along the median of the database according to a selected coordinate, can be used to build a kd-tree without pointers: store all data points in an array, search the median and place it at the center position of the array (round the index if there is no exact center position). Now place all other points at positions lower or higher than the center position of the array according to their relation to the median point in the selected coordinate. Selecting the median and rearranging the elements can be performed in a single step with the quick-median algorithm [7], [8]. Thus at each position in the array, only the data point, and integer for the coordinate along which this point splits the data set, needs to be stored.

The probably fastest implementation of the kd-tree uses an additional optimization based on the observation that the arrangement of the elements in the array according to the median placement leads to a very sparse and cache-inefficient access pattern of the array. By viewing the array as a balanced heap, the binary kd-tree can be stored in such a way that the root of the tree is in the first element, its two children are in the two subsequent element, and so on. An implementation of this optimization has been published by Jensen [1].

2.2 The vantage point tree

The vantage point tree (vp-tree) introduced by Yianilos [2] uses the same idea of hierarchic bisection of the data set. However instead of splitting along the coordinate values, at each level a specific vantage point is selected, and the points are partitioned according to the distance from this vantage point. Thus a sphere around the vantage point bisects the point set into a near and a far subset, the near set is stored in the left subtree, the far set is stored in the right subset. An example vp-tree can be seen in figure 2.

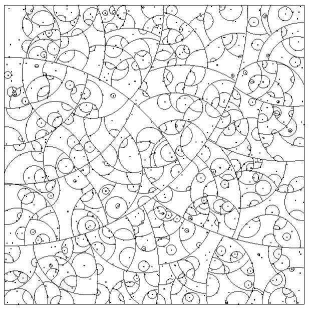

Fig. 2 Two dimensional example vp-tree decomposition (image from Yanilos' paper [3]).

The vantage point should be selected in such a way to avoid splitting the far set into disjoint regions. This can often be done by choosing vantage points near the corners of the point set to split [3].

The advantage of the vp-tree when compared to the kd-tree can be characterized as follows: in higher dimensions the partitioning with respect to the vantage points partitions space in all dimensions at once, and thus for non-uniform distributions of points, fast decisions can be performed for entire subtrees, if a vantage point and its near points are completely inside or outside the current search sphere. This is especially noticeable if point sets of lower dimensionality are embedded in higher dimensional spaces, and in this case large performance gains over the kd-tree have been observed.

Optimized forms of the vp-tree include additional subspace bounds and may employ buckets near the leaf level. A number of improvements of the vp-tree relating to high-dimensional settings, distribution adaptation and incremental searches have been described [9], [10], [11].

3 The rkd-tree

The observation from the vantage point tree that fast decisions can be performed if certain sub-sets of points are known to be contained in spheres of a known radius, can be applied to the kd-tree as well. For each split-point of the kd-tree we compute and store the *radius* **r** (hence the name **r**kd-Tree) of the sphere containing all points in its left and right subtrees. In figure 3, the spheres of four split-points at the same level of the tree are shown in red.

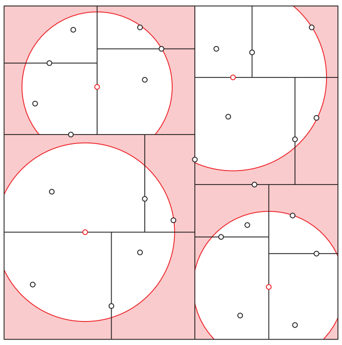

Fig. 3 Two dimensional example rkd-tree decomposition. The spheres around the nodes at one level of the tree are shown in red, the regions that are potentially excluded from rkd-tree searches (when compared to kd-tree searches) are shaded.

The search algorithm for the n-closest points to a given query point maintains the radius of the sphere around the query point containing the current best set of n closest points. In a normal kd-tree search this sphere is only classified with respect to the kd-tree split planes. In the rkd-tree we use this radius and the stored radius around each split-point to perform an additional exclusion test. This test comes nearly for free, since the distance from the current best set to the split-point needs to be computed in the standard kd-tree as well in order to test if the split-point needs to be included in the current best set. Thus the only work to perform is the evaluation of the triangle inequality of the radius of the split-point sphere and the radius of the current best set with respect to the distance of the split-point to the query point. If the triangle inequality does not hold, both sub-sets of the split point can be excluded from the search.

4 Evaluation

In order to perform a meaningful evaluation of the performance of the rkd-tree, we used an optimized C# implementation based on the heap/array optimizations by Jensen [1] for all three trees: kd-tree, vp-tree, and rkd-tree. Note that this leads to a highly efficient vp-tree implementation that has not been documented in literature up to now.

4.1 Artificial point sets

Our first test consisted of queries for the closest 100 points in dense random sets of 10^6 points of increasing dimension, using three different distance metrics. Figure 4 shows the relative timing of the kd-tree and the vp-tree when compared to the rkd-tree.

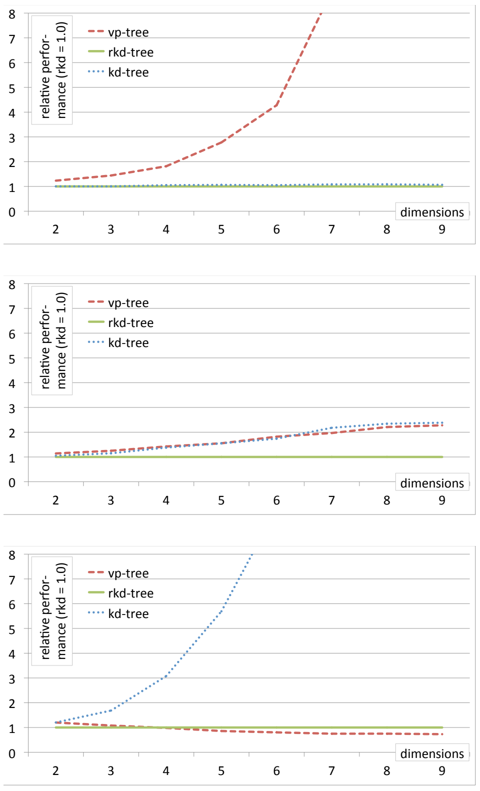

Fig. 4 Relative timings of the kd-tree and the vp-tree compared to the rkd-tree in queries for the 100 closest points in **dense** random point sets of 10^6 points of varying dimensionality for L^∞, L^2, and L^1 distance metric (top to bottom).

In queries with euclidean metric (L^2), the rkd-tree significantly outperforms both the kd-tree and the vp-tree in this evaluation. In the maximum metric (L^∞), the rkd-tree is minimally faster than the kd-tree are and both are a lot faster than the vp-tree. Only in queries using the manhattan distance (L^1) the rkd-tree is somewhat outperformed by the vp-tree.

The second test (figure 5) uses the same setup, except that point sets of lower dimension were embedded in higher dimensional spaces (the data set dimension was a third of the space dimension).

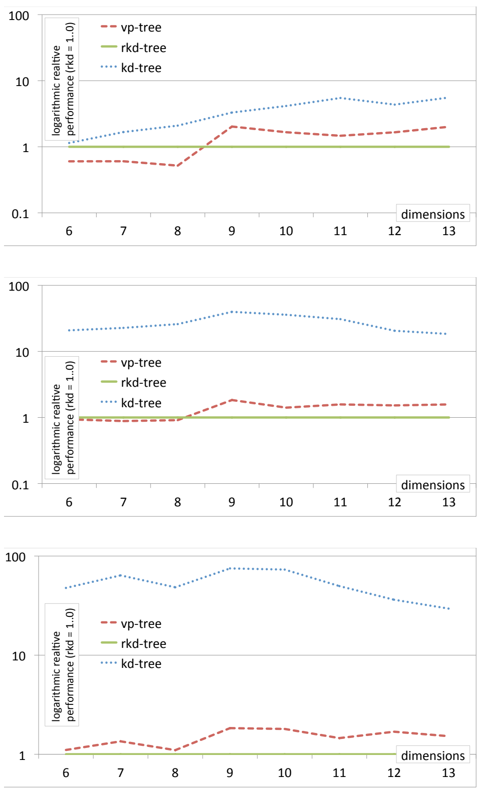

Fig. 5 Relative timings (on a logarithmic scale) of the kd-tree and the vp-tree compared to the rkd-tree in queries for the 100 closest points in **sparse** random point sets of 10^6 points for L^∞, L^2, and L^1 distance metric (top to bottom). The dimensionality of the data sets was a third of the dimensions of the space (shown on the x-axis) they were embedded in.

On these sparse data sets the rkd-tree is in most cases slightly faster than the vp-tree, and significantly faster than the kd-tree.

4.2 Real world point sets

Kd-trees are often used in photon mapping algorithms [1] to find photons that are close to an evaluation point. In order to create a more discriminate search it is possible to add the similarity of the normal vector as an additional search criterion. This can be accomplished by putting the photons with their normal vectors into a 6-D search space (three dimensions for the point location, three dimensions for the normal vector) with the normals scaled by a factor according to the desired importance. We have run a number of tests for photon data sets generated when illuminating a city block (see figure 6). The results show that as the importance of the normal is scaled up, the rkd-tree is increasingly faster than the kd-tree with speed ups ranging from 37% with low normal importance up to 113% for high normal importance. When compared to the vp-tree, the rkd-tree is in most cases slightly faster (about 10% to 12%), only in the extreme cases, where the normal importance dominates the search, it is slightly slower (up to 7%).

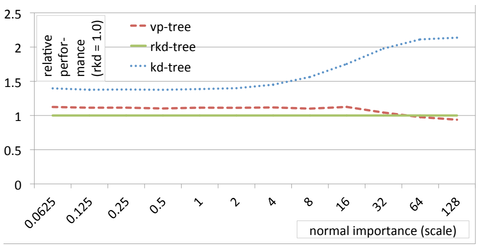

Fig. 6 Relative timings of the kd-tree and the vp-tree compared to the rkd-tree in queries for the 250 closest points with similar normals in a data set of 10^5 photons simulating the solar illumination of a city block of 250 by 250 meters. The normal importance is given by the scaling of the normal (in meters). Note that if the normal is scaled to be very small the results are nearly identical as if the search was performed without normals in 3D space.

5 Conclusion

We presented an improvement of the kd-tree for n-closest point searches that is based on the vp-tree. It mostly retains or exceeds the kd-tree speed in dense data sets, and mostly retains or exceeds the speed of the vp-tree in sparse data sets. Thus it delivers a stable performance regardless of the nature of the data set.

References

1. H. Jensen, Rendering Techniques **96**, 21 (1996)
2. P.N. Yanilos, Invited Talk: The Instituted of Defense Analyses (1990)
3. P.N. Yianilos, in *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1993), pp. 311–321
4. J.H. Friedmann, F. Baskett, J. Shustek, IEEE Transactions on Computers (1975)
5. J.H. Friedman, J.L. Bentley, R.A. Finkel, ACM Trans. Math. Softw. **3**(3), 209 (1977). DOI http://doi.acm.org/10.1145/355744.355745
6. J.L. Bentley, Commun. ACM **23**(4), 214 (1980). DOI http://doi.acm.org/10.1145/358841.358850
7. R. Sedgewick, Communications of the ACM **21**(10), 847 (1978)
8. R. Sedgewick, (1998)
9. S. Eastman, Information Systems **7**(2), 115 (1982)
10. B. Kim, S. Park, Pattern Analysis and Machine Intelligence, IEEE Transactions on **8**, 761 (1986)
11. A. Broder, Pattern Recognition **23**(1-2), 171 (1990)