Collision detection between point clouds using an efficient k-d tree implementation

Johannes Schauer, Andreas Nüchter

Informatics VII: Robotics and Telematics, Julius-Maximilians-University Würzburg, Am Hubland, Würzburg 97074, Germany

Abstract

Context: An important task in civil engineering is the detection of collisions of a 3D model with an environment representation. Existing methods using the structure gauge provide an insufficient measure because the model either rotates or because the trajectory makes tight turns through narrow passages. This is the case in either automotive assembly lines or in narrow train tunnels.

Objective: Given two point clouds, one of the *environment* and one of a *model* and a trajectory with six degrees of freedom along which the model moves through the environment, find all colliding points of the environment with the model within a certain clearance radius.

Method: This paper presents two collision detection (CD) methods called *kd-CD* and *kd-CD-simple* and two penetration depth (PD) calculation methods called *kd-PD* and *kd-PD-fast*. All four methods are based on searches in a k-d tree representation of the environment. The creation of the k-d tree, its search methods and other features will be explained in the scope of their use to detect collisions and calculate depths of penetration.

Results: The algorithms are benchmarked by moving the point cloud of a train wagon with 2.5 million points along the point cloud of a 1144 m long train track through a narrow tunnel with overall 18.92 million points. Points where the wagon collides with the tunnel wall are visually highlighted with their penetration depth. With a safety margin of 5 cm kd-PD-simple finds all colliding points on its trajectory which is sampled into 19392 positions in 77 s on a standard desktop machine of 1.6 GHz.

Email addresses: `johannes.schauer@stud-mail.uni-wuerzburg.de` (Johannes Schauer), `andreas@nuechti.de` (Andreas Nüchter)

URL: `http://www.nuechti.de` (Andreas Nüchter)

Conclusion: The presented methods for collision detection and penetration depth calculation are shown to solve problems for which the structure gauge is an insufficient measure. The underlying k-d tree is shown to be an effective data structure for the required look-up operations.

Keywords: Collision Detection, Interference Detection, K-D Tree, Kinematic Laser Scanning, 3D Point Clouds

---

1. Introduction and problem formulation

The minimum clearance outline or *structure gauge* has an important place in the planning of rail and automotive infrastructure as well as for factory assembly lines [1]. It is the swept volume of the minimum cross section that must be kept free of any obstacles. Measuring the structure gauge of railroad and motorway tunnels, bridges and production lines is a simple way to calculate whether vehicles, their cargo or arbitrary objects can pass through them. The structure gauge is an exact measure as long as the moving object travels along a straight line and does not rotate. But if the trajectory is not straight or rotation is involved, then the structure gauge can only serve as a rough estimation which becomes more imprecise the shorter the turn radius or the larger the rotation of the moving object. Normal railroads and rural motorways usually are constructed with long turn radii and large safety margins, so the structure gauge is a sufficient measure to determine whether a vehicle can pass along a route. But there exist many examples where the structure gauge is an insufficient measure:

- transportation of exceptionally long, rigid cargo along motorways and railroads
- turns in very narrow tunnels, bridges or other passages
- street turns with a very small turn radius (for example in urban environments)
- rotating objects along production lines

The collision detection method presented in this paper solves this problem but can also be applied to general collision detection tasks. The difference to most other collision detection algorithms is that this method is purely point based and does not require to calculate a solid 3D mesh representation.

This method was first applied by the authors to find collisions in an automotive production line which involved sharp turns and rotations of the car body but the respective paper focuses on the techniques to register the environment [1]. In the

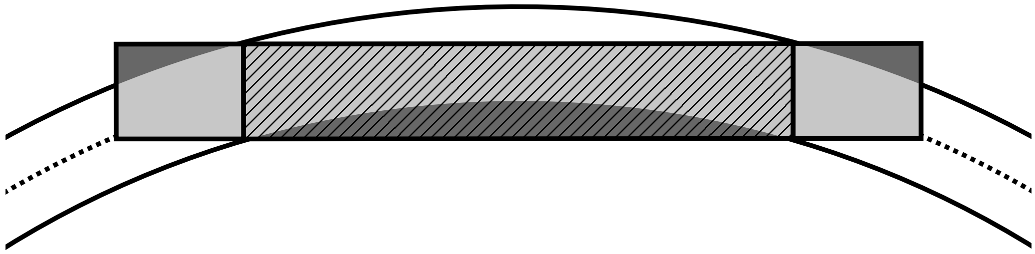


Figure 1: Top view of the train wagon (in dark and light gray) and its curved loading gauge as it passes through a turn. The dark gray areas mark the volumes of the train wagon outside of its loading gauge. The striped volume indicates the volume of the train wagon between its two bogies. The dotted line indicates the wagon's trajectory.

following, the same method with some further improvements will be applied to a train moving through a very narrow tunnel where a structure gauge based approach does not sufice to find collisions but where there will be collisions in reality because of the turn the tunnel makes.

A similar measure to the *structure gauge* is the *loading gauge* which is the swept volume of the cross section of a train wagon moved along a track. The difference between the two is the engineering tolerance or *clearance*. The structure gauges along a track together with the maximum loading gauge determine whether or not a train with certain cargo can go along a given route or how much space around new tracks has to be kept clear and is subject to a number of decades old standards and regulations [2].

If the "track transition curve" at the start and the end of most turns is ignored, then turns of train tracks always represent circle segments (i.e. circular arcs) [3]. Since the rotation centers of the two bogies of a train wagon both stay in the exact center between the train tracks, the part of the train wagon connecting the bogies will form the line segment of a secant cutting the circle segment of the track. Thus, the parts of the wagon between the bogies in the inside of the turn will take more space of the structure gauge within a turn compared to when the train wagon travels along straight tracks. Similarly, the parts of the train wagon on both ends outside of the bogies will take additional space as well. Figure 1 visualizes the problem. The curvature represents the loading gauge of the train wagon in gray. The dark gray areas represent the volume of the train wagon which is outside of its loading gauge during the turn. The amount of needed additional space is depending on the turn radius. To address the problem, there exist different regulations for structure gauge sizes depending on the turn radius [4].

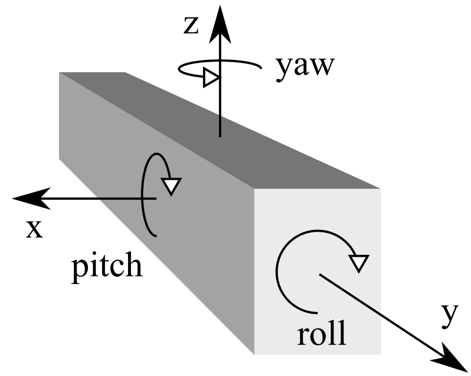


Figure 2: The train wagon is oriented and moves along the y-axis.

The algorithms that will be presented in the following requires three objects as input: The first input is the pointcloud of the *environment*. In the example presented throughout the paper, it was collected by driving a Optech Lynx Mobile Mapper along the train tracks but can also be acquired using the methods presented in [1]. The second input is a point cloud of the *model*. Here, it was acquired by taking seven terrestrial 3D scans of a real train wagon with a Riegl VZ-400 laser scanner and then registering them using *3DTK – The 3D Toolkit* [5]. The third input is the *trajectory* of the train tracks.

The goal is to determine which points of the environment collide with the model on its path, given a certain safety margin (the minimal allowed clearance) and how deep any colliding points of the environment penetrate the model. To this end a k-d tree of the environment is created, the model is moved through it along its trajectory and a k-d tree search is performed around the points of the model to find colliding points and their penetration distance.

The contributions of this paper are summarized as:

- a method to perform collision detection of a single arbitrary (and deformable) point cloud (the model) with a static environment in two variants (kd-CD and kd-CD-simple)
- two methods to calculate penetration depth of the model with the environment (kd-PD and kd-PD-fast)
- a highly optimized k-d tree implementation and query functions to perform collision detection

A right handed coordinate system will be assumed in this paper. Figure 2 shows the local coordinate system of the train wagon. The z-axis is the up vector and the train wagon is moved along its y-axis. The wagon is centered such that its center of mass is in the origin of the coordinate system. This is important for calculating rotations and penetration depths.

The remainder of the paper is organized as follows: The next section covers related work to the one presented in this paper. Section three presents the k-d tree data structures and algorithms. Section four and five present our methods for collision detection and penetration depth calculation using the k-d tree, respectively. Section six show benchmark results while section seven concludes this paper.

2. Related Work

Collision detection, which is also called interference detection or intersection searching, is a well studied topic in computer graphics [6, 7, 8, 9, 10] because of its importance for dynamic computer animation and virtual reality applications [11, 12, 13]. On the other hand, their work is limited to collision detection between geometric shapes and polygonal meshes whereas most sensor data is acquired as point clouds. While collision detection is also relevant for motion planning in the field of robotics, it is a less studied problem there. Collision detection between point clouds was for example researched by Klein and Zachmann [14] who use the implicit surface created by a point cloud to calculate intersections. Another example is the recent work by Hermann et al. [15] who use voxels to check for spatial occupancy for robot motion planning.

Existing techniques make use of very similar approaches. One method is to apply a spatial hierarchical partitioning of the input geometry using octrees [16, 17], AABB-trees [18], BSP-trees [19] or k-d trees [20] . Other solutions apply regular partitioning using voxels [21, 15, 22]. The goal of any partitioning is to be able to quickly search and check only the relevant geometries in the same or neighboring cells. The method presented in this paper will make use of a hierarchical k-d tree for the environment in combination with a regular partitioning of the model into a grid of bounding spheres.

Another method is to use hierarchies of bounding volumes like spheres [23], axis aligned bounding boxes [24], oriented bounding boxes [25] or discrete oriented polytopes [26]. Optimizing the regular grid that was generated for the model into a hierarchical structure will be left for future work.

Collision detection methods can be divided in those for static and deformable objects [27, 28]. While the method presented in this paper does not easily allow changes in the environment because that does require a recalculation of its k-d tree, arbitrary changes in the point cloud of the model are possible without any performance impacts.

Another classification is whether the algorithm easily allows multiple moving objects. Using a brute-force approach such algorithms have a runtime of $O(n^2)$ for n objects because every possible pair of objects is checked for collisions. Modern approaches like the I-COLLIDE system [29] use a "sweep and prune" approach to minimize the amount of necessary checks. Another approach is to dynamically adjust the search tree to account for object movements [30]. The method in this paper does not handle multiple moving models.

Calculating the penetration depth of one object into another is important to calculate the force of collisions and respond accordingly in virtual reality applications [11]. It is also important for visualization purposes, to differently highlight objects reaching into a safety margin with an indication of how much they violate the constraint. This application was shown in prior work on this topic by the authors of this paper [1].

The k-d tree implementation this work bears similarities to R+-trees [31] insofar it recalculates a new bounding box for each child node. In contrast to R+-trees, the k-d tree implementation presented here does not make efforts to create a balanced tree. In [32] our k-d tree implementation was benchmarked against three nearest-neighbor search libraries based on the k-d tree data structure: ANN [33], libnabo [34] and FLANN [35] and came out amongst the fastest implementations.

3. Data Structures for efficient collision detection and depth of penetration calculation

In this section our highly-optimized k-d tree implementation is presented. It is implemented in 3DTK [5] in C++. It currently implements multiple search functions, can be parameterized to be used with 3D point data of different precision and container type, allows to present search results as pointers, array indices or as 3D coordinate data and allows parallel execution through OpenMP. Its correctness has been verified by a test suite which combines brute force implementations of the search functions (test all points for satisfaction of the search criterea) against the result of a search in the k-d tree.

The recursive function `_FixedRangeSearch` which returns a STL vector of all points within a certain radius r around a coordinate P will be used as an example throughout this section. In the following code examples all class members which are not directly useful for the execution of `_FixedRangeSearch` are omitted for brevity.

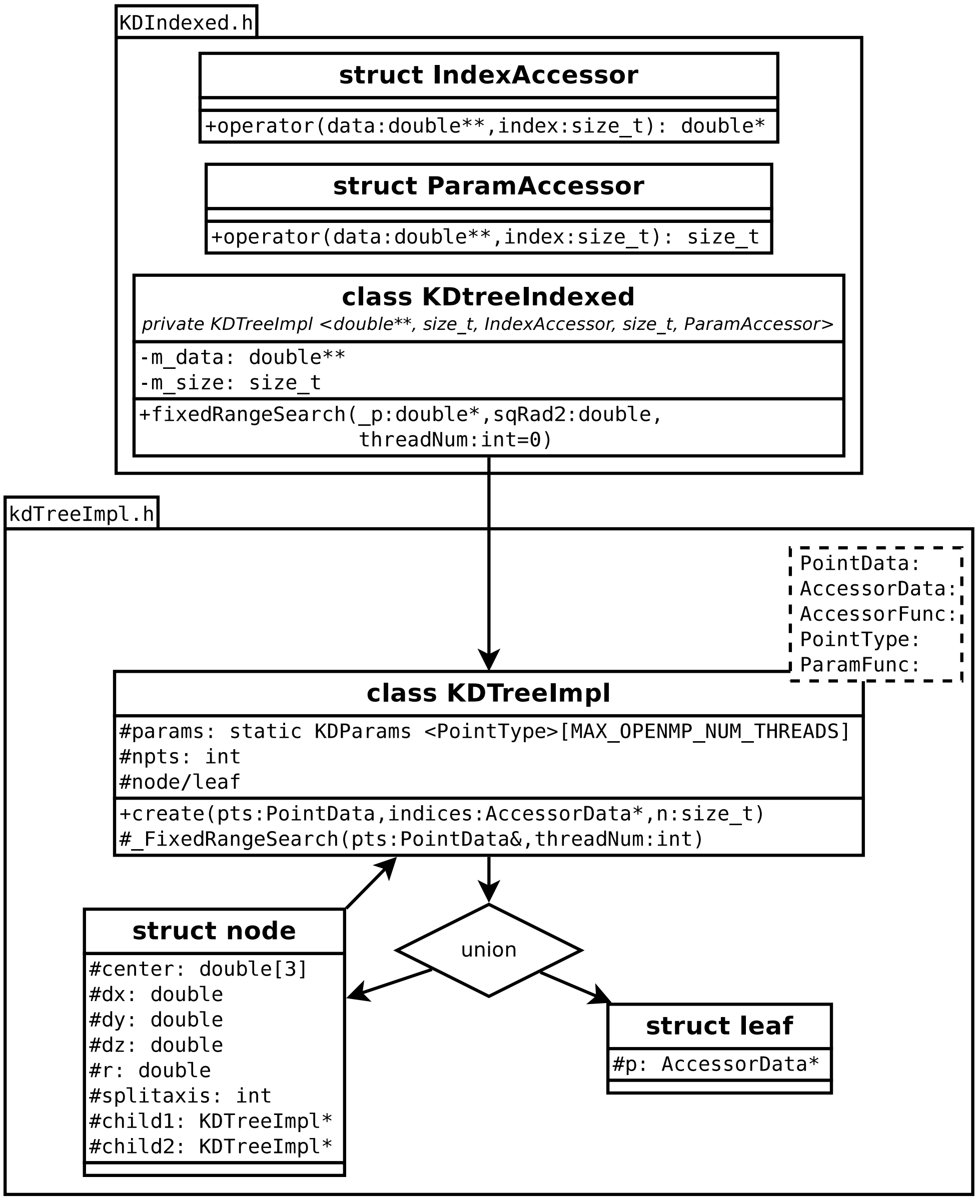


Figure 3: Boxes are in UML, relationships (arrows) are not. `KDTreeImpl` is templated by `KDtreeIndexed` with the parameters listed in the comment. The `node` and `leaf` structure are a C++ `union`. The `params` member of `KDTreeImpl` is static. UML packages are used to indicate file membership and to group for readability. Only the `fixedRangeSearch` search function and its recursive counterpart `_FixedRangeSearch` are listed for brevity.

For an overview, consider figure 3. The template class `KDTreeImpl` provides the implementation of search functions and at the same time represents an inner node or a leaf node of the k-d tree. Multiple classes instantiate `KDTreeImpl`, one of them being `KDtreeIndexed` which is of particular use for the collision detection method in this paper. The classes and functions seen in figure 3 will be explained in more detail in the following sub-sections.

The general operation of the search functions will be presented by using the function `fixedRangeSearch` as an example. The function is implemented in the class `KDTreeIndexed`. It sets up the `KDParams` structure with the search parameters and then calls the recursive function `_FixedRangeSearch` (notice the leading underscore) implemented in `KDTreeImpl`. The function `_FixedRangeSearch` in turn implements the actual search operations.

3.1. Tree data structure

```
1 template<class PointData, class AccessorData, class AccessorFunc,
2 class PointType, class ParamFunc> class KDtreeImpl {
3   public:
4     void create(PointData, AccessorData *, int);
5   protected:
6     static KDParams<PointType> params[MAX_OPENMP_NUM_THREADS];
7     int npts; // equal zero for inner nodes, otherwise leaf
8     union {
9       struct {
10        double center[3];
11        double dx, dy, dz;
12        int splitaxis;
13        KDtree* child1, *child2;
14      } node;
15      struct { AccessorData* p; } leaf ;
16    };
17    void _FixedRangeSearch(const PointData&, int);
18 };
19 template<class T> class KDParams {
20   public:
21    double maxdist_d2;
22    double *p;
23    vector<T> range_neighbors;
24 }
```

Listing 1: k-d tree implementation classes

Listing 1 shows an excerpt from the template class `KDtreeImpl`. Each instance of the class represents an inner or leaf node in the k-d tree.

The public `create` function in line 4 recursively creates a k-d tree by splitting the points it received as an argument into two, creating two new instances of `KDtreeImpl` and calling their `create` function with one of the new point sets, respectively. The inner working of the `create` function is explained in section 3.2.

The static member `params` in line 6 is set once for every new search in the k-d tree. It avoids having to pass the search parameters for each recursive function call and thus reduces the size of required operations on the stack. As it is a static member, it will only be stored in memory once, i.e., hardware cache friendly. The `KDParams` class in this shortened excerpt stores the point around which to search `p`, the squared search radius `maxdist_d2` and the search result vector `range_neighbors`. Since it is possible to carry out searches in the same k-d tree in parallel, an array of size `MAX_OPENMP_NUM_THREADS` exists.

The member `npts` in line 7 stores the number of points this node contains. If this value is non-zero, the node is a leaf node. Otherwise, the node is an inner node. Depending on the node type, a union structure in line 8 stores data about the node. Inner nodes store their center coordinate (line 10), the node size (line 11), the coordinate axis by which the node is split (line 12) and pointers to the two children the node is split into (line 13). Leaf nodes store a pointer `p` to an array representing the contained points (line 15).

3.2. Building the k-d tree

A k-d tree is created by instantiating `KDtreeImpl` and calling its `create` method with the points one wants to fill the k-d tree with. The `create` method will then recursively instantiate new `KDtreeImpl` child nodes until all points are distributed into leaf nodes. The `create` method is shown as an abbreviated excerpt in listing 2 is explained in more detail in the following.

The first check in line 2 decides whether the current node is an intermediate node or a leaf node. If the number of points passed to the `create` function is less than or equal to 10 then this node will become a leaf node storing all points it is given and recursion stops. Otherwise the node is an inner node. This is recorded in the `npts` member in line 8. The number 10 is chosen as the bucket size because of run-time evaluations done in [36] (see figure 5 in that paper).

The clipped lines 9-11 calculate an axis aligned bounding box for the points the function is given. The bounding box is represented as its center point and its half length, width and height. Thus, the values `node.dx`, `node.dy` and `node.dz` store the distance from the center to the sides of the bounding box. The axis by which to split the bounding box into two is found in lines 12-18. The split is done by determining the longest axis and splitting the bounding box in half by that axis.

Lines 19-23 partition the points the `create` function is given. To reduce the amount of required copies, the original array with points is reused and split into half.

```
1 KDtreeImpl::create(PointData pts, AccessorData *indices, int n) {
2   if (n > 0 && n <= 10) { // Leaf nodes, copy data
3     npts = n;
4     leaf.p = new AccessorData[n];
5     for (int i = 0; i < n; ++i) leaf.p[i] = indices[i];
6     return;
7   }
8   npts = 0; // inner node
9   // finding bounding box
10  // node.center, node.dx, node.dy, node.dz
11  [...]
12  // calculate longest axis
13  if (node.dx > node.dy)
14    if (node.dx > node.dz) node.splitaxis = 0;
15    else node.splitaxis = 2;
16  else
17    if (node.dy > node.dz) node.splitaxis = 1;
18    else node.splitaxis = 2;
19  // distributing data to fields left and right for the
20  // following nodes according to splitval
21  double splitval = node.center[node.splitaxis];
22  AccessorData *left;
23  [...]
24  // creation of subtrees
25  node.child1 = new KDtreeImpl();
26  node.child1.create(pts, indices, left-indices);
27  node.child2 = new KDtreeImpl();
28  node.child2.create(pts, left, n-(left-indices));
29 }
```

Listing 2: *k*-d tree creation

Only points which happened to be on the wrong side are swapped with wrong points on the other side. On average this halves the amount of required copy operations. In the end, `indices` will point to the left hand side half of the original array while `left` will point to the right hand side half of the array. The last lines 24-28 instantiate two new `KDtreeImpl` objects and call their `create` function with the respective, sorted half of the original input data.

3.3. k-d tree layout

The `create` function explained in section 3.2 will result in a partitioning of the input points as shown in figure 4 which shows a simplified two-dimensional representation of the input points and the resulting tree structure in memory. In contrast to a classical *k*-d tree, the search volume of child nodes is reduced by recalculating a bounding box for the enclosed points. This technique is similar to how R-trees operate and helps to create a tighter boundary for the enclosed points which in turn results in performance improvements during look-ups. This is because restricting

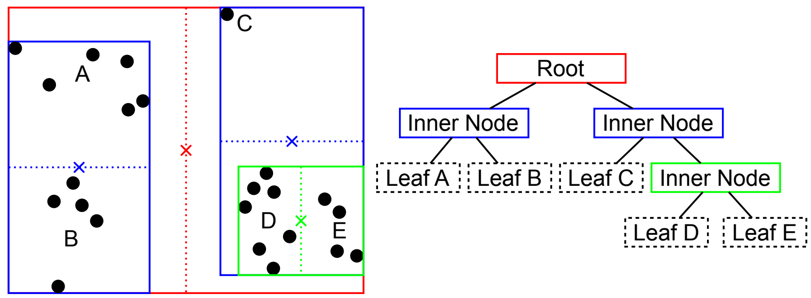


Figure 4: Left: 23 points (black circles) and the bounding boxes (solid lines), their centers (crosses) and their split axis (dotted lines) of the 2-dimensional k-d tree created from them. The letters identify the created groups of points per leaf node. **Right:** The tree representation of the created 2D k-d tree. The color of the solid boxes corresponds to the bounding boxes in the left figure. Boxes with dotted outlines are leaf nodes. The names of the leaf nodes correspond to the letters in the left figure.

the bounding volume of child nodes to a new bounding rectangle allows to abort a search quickly instead of having to search the k-d tree until leaf nodes are reached and inspected.

Considering figure 4, the `create` function is first called with all 23 points as an argument. Since $23 > 10$, a new inner node will be created by calculating the node center and its bounding box (in red). The bounding box is wider than it is high so the points will be partitioned by a vertical axis through the bounding box center. Two new `KDtreeImpl` instances are created for each side and get passed 11 and 12 points, respectively. Since both values are greater than 10 again, new inner nodes will be created with their bounding boxes shown in blue. The following iteration will then result in two leaf nodes on the left hand side (6 points in the upper region and 5 points in the lower region) and one leaf node on the right hand side with one point. One last iteration over the remaining 11 points on the right hand side will create two last child nodes. Leaf nodes do not require a bounding box because when they are encountered during a k-d tree search, all the points they contain are checked and no further recursion has to be done.

3.4. Searching the k-d tree

Spacial search in point clouds are parameterized by two properties: the location (where to search for results) and the subject (what to return). The following five search areas are implemented by 3DTK:

(a) radius r around a point P_1

(b) radius r around an infinite line defined by a point P_1 and a direction vector v

(c) radius r around an infinite ray defined by P_1 and v

(d) radius r along a finite line segment defined by points P_1 and P_2 and

(e) inside an axis aligned bounding box defined by P_1 and P_2 as the corners with minimum and maximum coordinate values, respectively

Additional search volumes that can be added in the future would be oriented bounding boxes, cylinders or general polytopes. In most volumes, it is possible to perform searches for the following result types:

(1) the point closest to P_1

(2) the k points closest to P_1

(3) all points within the search volume

(4) the point closest to the given line, ray or line segment

(5) the k closest points to the given line, ray or line segment

After eliminating the inapplicable combinations, one ends up with 19 meaningful search functions. A full list is omitted for brevity. For example, the common nearest-neighbor search (NNS) is searching for the closest point to P_1 (1) in a radius r around a point P_1 (a). For the collision detection method presented in this paper, the following four functions are needed:

- `FindClosest`: closest point to a coordinate: (a) and (1)
- `fixedRangeSearch` all points around a coordinate: (a) and (3)
- `segmentSearch_1NearestPoint` closest point to P_1 in a line segment: (d) and (1)
- `segmentSearch_all` all points around a line segment: (d) and (3)

3.5. _fixedRangeSearch

All recursive search functions are divided into three functional parts. Firstly, the node is checked whether it is an inner node or a leaf node. If it is a leaf node, then all points the node contains are checked for satisfiability of the search criteria and the function returns. The second part is reached if the node is an inner node and thus the first part did not cause the function to return. In that case, a check is done whether the node can possibly contain parts of the result. If not, then the function returns. Otherwise, thirdly, the search recurses into one or both child nodes.

```
1 void KDtreeImpl::_FixedRangeSearch(const PointData& pts,
2 int threadNum) {
3   AccessorFunc point; ParamFunc pointparam;
4   if (npts) { // node is leaf
5     for (int i = 0; i < npts; i++) {
6       double myd2 = Dist2(params[threadNum].p,
7                           point(pts, leaf.p[i]));
8       if (myd2 < params[threadNum].maxdist_d2)
9         params[threadNum].range_neighbors.push_back(
10          pointparam(pts, leaf.p[i]));
11    }
12    return;
13  }
14  // quick test whether subtree has to be searched
15  double approx_dist_bbox =
16    max(max(fabs(params[threadNum].p[0]-node.center[0])-node.dx,
17    fabs(params[threadNum].p[1]-node.center[1])-node.dy),
18    fabs(params[threadNum].p[2]-node.center[2])-node.dz);
19  if (approx_dist_bbox >= 0 && sqr(approx_dist_bbox)
20     >= params[threadNum].maxdist_d2) return;
21  // recursive case
22  double myd = node.center[node.splitaxis]
23               - params[threadNum].p[node.splitaxis];
24  if (myd >= 0.0f) {
25    node.child1->_FixedRangeSearch(pts, threadNum);
26    if (sqr(myd) < params[threadNum].maxdist_d2)
27      node.child2->_FixedRangeSearch(pts, threadNum);
28  } else {
29    node.child2->_FixedRangeSearch(pts, threadNum);
30    if (sqr(myd) < params[threadNum].maxdist_d2)
31      node.child1->_FixedRangeSearch(pts, threadNum);
32  }
33 }
```

Listing 3: k-d tree search

Consider listing 3 which shows the function `_FixedRangeSearch` as implemented in the `KDtreeImpl` class. It fills the result vector in the `KDParams` static member with all points in the k-d tree which lie around a certain squared radius `maxdist_d2` around a point `p`.

The parameterized functions of type `IndexAccessor` and `ParamAccessor` in line 3 are used to return coordinate data or data of the type stored in the results vector for each point in the leaf node, respectively. They do not pose a performance overhead as they are inlined by the compiler.

In case the node is found to be a leaf in line 4, all points in the leaf are checked whether their squared distance `myd2` to P is less than r. If they do, then they are appended to the result vector.

After all points in the leaf node have been checked, the function returns. If the node is not a leaf node but an inner node, then the next part from line 15-20 checks whether further recursion into the child nodes of this node is required. This check whether to abort will be outlined in the next subsection 3.6.

The last part of each search function in lines 22-32 recurses into the child nodes. First, a check for the point's position relative to the split axis of the current node (as calculated in line 22) decides which child node to recurse first. Whether or not the other child node is recursed into as well depends on whether the bounding cube of the search radius around P can possibly extend into the other child as well or not.

3.6. Quick check whether to abort

A heuristic was developed that allows a quick check whether or not to continue searching further down the current branch of the k-d tree. Lines 15-18 in listing 3 implement this check in C++. This code compiles to only 16 SSE2 instructions and requires no branching operations like a trivial check otherwise would.

The algorithm works by calculating a value d_P which is then compared to the search radius to decide whether or not to abort the search in the k-d tree. In the following formula, P is the three dimensional coordinate of the point around which the search is to be done. The current node of the k-d tree is parameterized by its center coordinate C and its axis aligned bounding box size $2d_x$, $2d_y$ and $2d_z$.

$$\begin{aligned} d_P = \max(&|P_x - C_x| - d_x, \\ &|P_y - C_y| - d_y, \\ &|P_z - C_z| - d_z) \end{aligned} \tag{1}$$

In words, suppose the six sides of the node's axis aligned bounding box form six axis aligned planes: each plane being the infinite extension of the six sides of the node's bounding box. Opposing sides of the node's bounding form pairs of parallel planes. Three of these plane pairs are created, one pair along each dimension. Then the distance of P to the closest plane of each pair of planes is found. If P is between a pair of planes, then its distance is represented as a negative value. Then the maximum distance of the resulting three distance values is taken (one for each dimension). If the maximum value d_P is negative, then all three coordinate values of P must lie inside the current node's bounding box and the search has to recurse into one or both child nodes. If the maximum value is positive and larger than the search distance, then the current node cannot contain any results and the function returns without recursing deeper into the tree.

The heuristic can easily be visualized in two dimensions by considering figure 6. Instead of a bounding box, a bounding rectangle is shown in yellow. Instead of

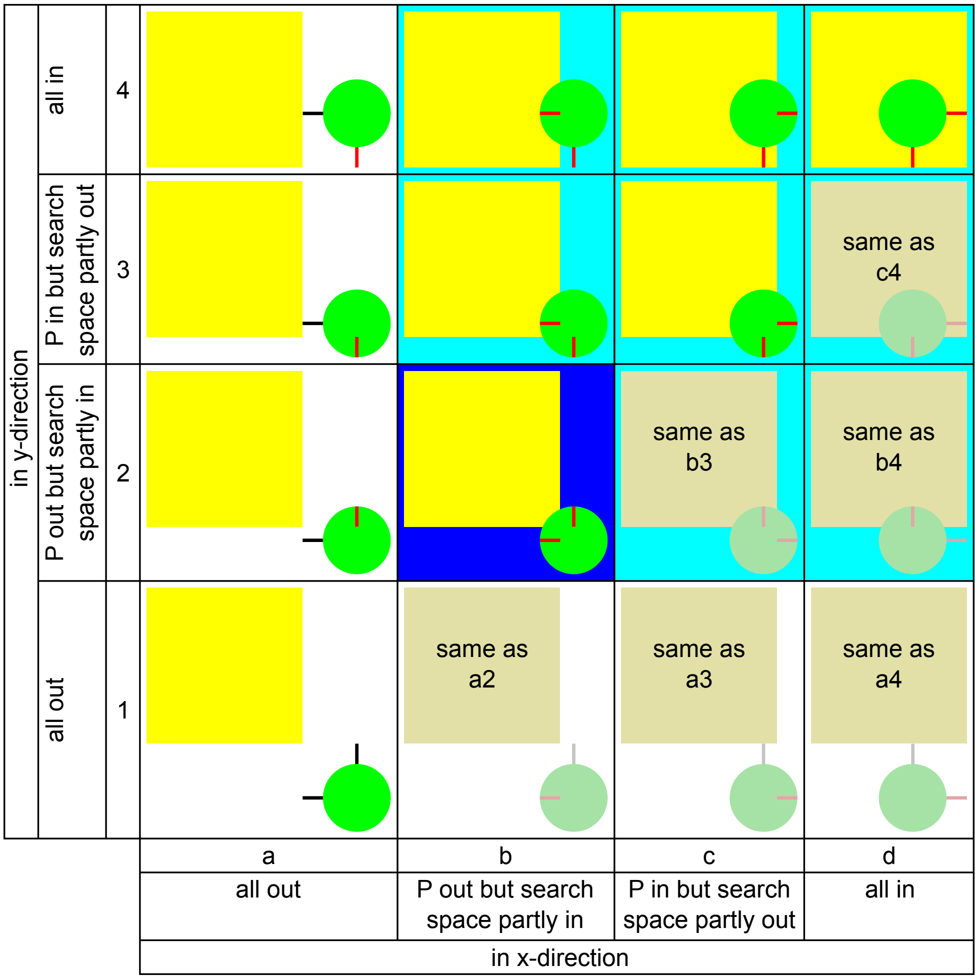


Figure 5: A two-dimensional overview of all possible locations a circular search radius (green) can have relative to the axis aligned bounding rectangle (yellow) of a two-dimensional k-d tree, ignoring rotations and mirroring. Each column represents a different horizontal position of the search radius relative to the bounding rectangle while each row represents a different vertical position. The lower-right triangle is faded out because it mirrors the upper left triangle along the diagonal. The black and red lines represent the positive and negative, respectively, distance from the search radius to the linear extension of the closest side of the bounding rectangle. The dark and light blue cells mark those positions in which parts of the search radius are found to lie in the bounding rectangle. In these cases, the search is not aborted as the search results might lie within the bounding box. In the other cases (cells with a white background) the search is aborted. The dark blue cell (b2) marks the case where this conclusion might lead to a false positive. See figure 6 for a more detailed overview.

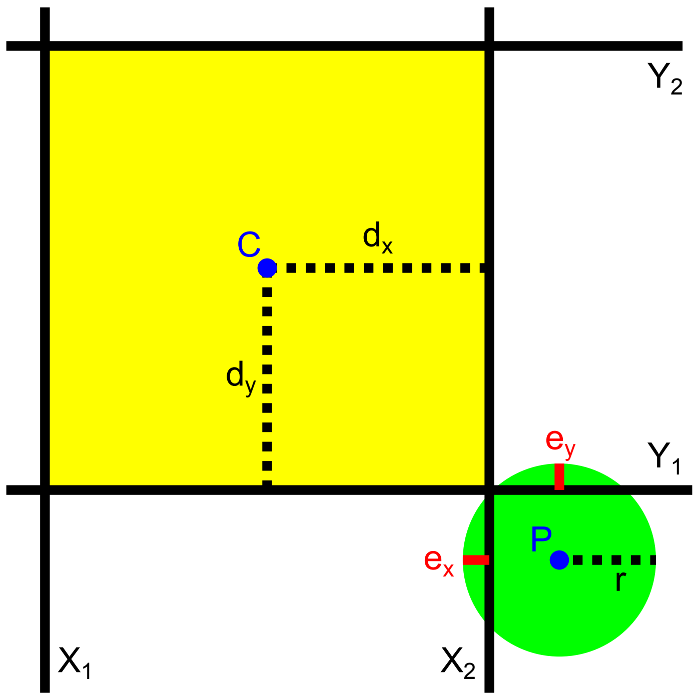


Figure 6: A close-up of cell b2 in figure 5. It shows the search radius (green) in a position which visualizes the false positive which will find the search radius to be intersecting with the axis aligned bounding rectangle (yellow) while there is no intersection in practice. Furthermore it shows the center of the bounding rectangle C, its size d_x and d_y, the center of the search radius P and its radius r as well as the linear extensions of the sides of the bounding rectangle X_1, X_2, Y_1 and Y_2. The distance e_x calculates as $|P_x - C_x| - d_x - r$. Since the result is negative, the line is colored in red. Similarly, e_y is calculated as $|P_y - C_y| - d_y - r$.

axis aligned bounding planes, axis aligned lines are shown in black, solid lines. This two-dimensional representation is used to create a matrix of all possible locations of the search volume relative to the bounding box in figure 5. The search is aborted in all cases displayed in cells with a white background.

Figure 6 also visualizes the point where this check is not precise and generates a false positive (also shown with a dark blue background in figure 5). Since only the bounding cube of the search radius r around P is concerned, it can happen that both bounding cubes intersect while the actual search sphere does not intersect. In this case, the check will not abort the recursion even though no result can possibly be found in the current node in this situation. This inexactness is not a problem for values of r which are of similar order of magnitude as leaf node sizes in the search area. In that case, the overhead of searching for matching points in the few leaf nodes that are wrongly classified is far less than the overhead that is created by a more expensive but exact check which requires branching. A similar enhancement to sphere/box intersection checks by replacing branching with the max operator is shown in [37].

If the search radius r grows bigger, then it might be worth to add a second, more exact check after the quick inexact check. This is done for our k-d tree search functions around line segments. While inexact, checking whether parts of a node's bounding sphere intersect with the line segment's bounding sphere first, before doing an exact check, increased the runtime by two to three orders of magnitude. It is up to further research whether it is worthwhile to develop a more clever method which is able to decide for the best check to abort in each situation.

3.7. Subclassing the k-d tree

While the class `KDtreeImpl` contains the algorithms to build and search a k-d tree, it needs to be subclassed by a class that specifies the parameters of `KDtreeImpl`, provides a frontend for the search functions and which fills the parameter container `KDParams` with the correct values.

Parametrization of the `KDTreeImpl` class allows to access coordinate data of different precision and container type through the `PointData` parameter. `AccessorData` allows different ways to access this data (through indices or pointers) while the `AccessorFunc` allows different ways of retrieving coordinate data with double precision from an array of `PointData` elements through an index given by the `AccessorData` type. The `PointType` parameter also governs how point data is stored in the shared parameter container `KDParams`. The `ParamAccessor` returns data of type `PointType` from the `PointData` type data array, given an index of type `AccessorData`.

This type of parameterization allows different use cases for the k-d tree. Originally, coordinate data was stored as pointers to three-tuple double arrays. This variant stores the data in the indices array, therefore having the identity function for `AccessorFunc` and `ParamFunc` and have `Void` as the `PointData` parameter. Later, support for the `DataXYZ` type was added which stores point data and attributes in a struct.

3.8. An indexing k-d tree

```
struct IndexAccessor {
  inline double *operator() (double** data, size_t index) {
    return data[index];
  }
};
struct ParamAccessor {
  inline size_t operator() (double** data, size_t index) {
    return index;
  }
};
class KDtreeIndexed : private KDTreeImpl<double**,
size_t, IndexAccessor, size_t, ParamAccessor> {
  public:  vector<size_t> fixedRangeSearch(double *, double);
  private: double **m_data;
}
```

Listing 4: An indexed k-d tree variant

For collision detection, we make use of the indexing functionality of `KDtreeImpl`. Data and indices are passed to the k-d tree during creation and the search functions return individual indices or vectors of indices. This is useful to quickly calculate a partitioning of the points into colliding and non-colliding points without having to perform pointer arithmetic and relying on a certain layout of the point data in memory. Returning the indices of a range search allows to quickly update boolean collision values in a second vector. As `IndexAccessor` and `ParamAccessor` are inlined by the compiler, they do not lead to a performance degradation.

Consider listing 5. The constructor of `KDtreeIndexed` (line 1) simply creates the underlying k-d tree by supplying it with the given point values and an indexing array (line 3). The function `FixedRangeSearch` fills the `KDParams` structure with info about the desired point P and search radius r in lines 7 and 8 and then calls the recursive search function that is implemented by `KDtreeImpl` in line 9. The search function saves its result in the `KDParams` structure, so they are copied to the final result vector in lines 10-14.

```
KDtreeIndexed::KDtreeIndexed(double **pts, size_t n) {
  m_data = pts;
  create(pts, prepareTempIndices(n), n);
}
vector<size_t> KDtreeIndexed::FixedRangeSearch(double *p,
double maxdist2, int threadNum) {
  params[threadNum].maxdist_d2 = maxdist2;
  params[threadNum].p = p;
  _FixedRangeSearch(m_data, threadNum);
  vector<size_t> result;
  for (auto it : params[threadNum].range_neighbors) {
#pragma omp critical
    result.push_back(*it);
  }
  return result;
}
```

Listing 5: Searching an indexed k-d tree

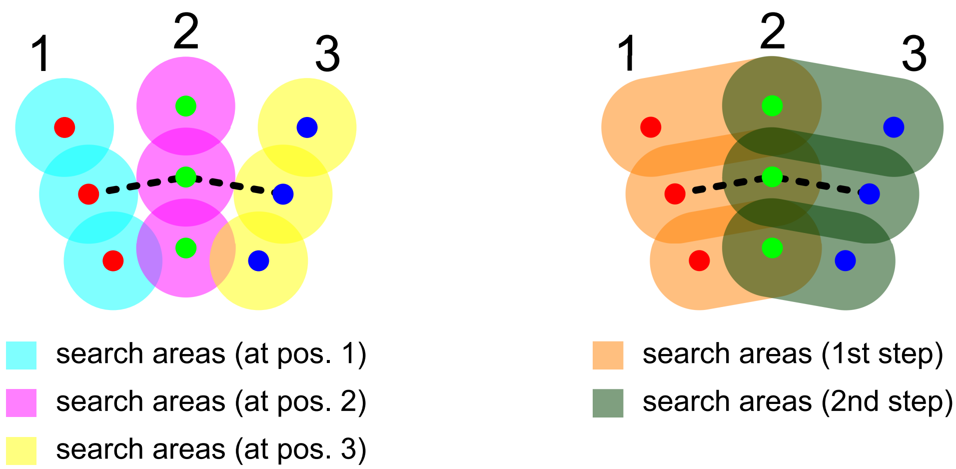
4. Collision detection

Two variants of collision detection are implemented using the k-d tree. One variant, called kd-CD-simple, is based on a range search around each point of the model using `FixedRangeSearch` and the other, called kd-CD, is based on a segment search between two subsequent points of the model on its trajectory using `segmentSearch_all`. In both variants, the model is moved along its trajectory and a range or segment k-d tree search with radius r is performed at each position.

When points are found to be colliding, then this information is saved in a separate boolean vector which stores for each point in the environment whether it ever collided with the model on its trajectory or not. The search radius r determines the precision of both algorithms. The smaller the search radius, the more precise the collision detection is. For smaller search radii, the model has to be sampled dense enough to not leave any unoccupied volume. The search radius r is the required "safety distance" between the model and the environment within which no point of the environment must lie. At the end, the collision information from the boolean vector is used to partition the environment into colliding and non-colliding points.

4.1. kd-CD-simple

In this variant, on each position of the model on its trajectory, a fixed range search using `FixedRangeSearch` is done around each point of the model. All points of the environment that are found to be within range r of any point of the model at any position on its trajectory are updated to be colliding. The performance of



(a) kd-cd-simple: with $T = 3$ points on the trajectory and $M = 3$ points of the model, $M \times T = 9$ `FixedRangeSearch` operations have to be carried out.

(b) kd-cd: with $T = 3$ points on the trajectory and $M = 3$ points of the model, $M \times (T - 1) = 6$ `segmentSearch_all` operations have to be carried out.

Figure 7: The two collision detection variants in two dimensions. A model consisting of three co-linear points is moved through the environment along a trajectory (dashed line) with three positions (indicated by numbers at the top). The first position of the three points of the model is marked with red dots, the second position of the model with green and the third position with blue dots. The area that is searched for collisions with the environment is indicated by the transparent colored areas.

kd-CD-simple is improved by sampling the model in a way such that the search radii around its points overlap in the desired amount.

Figure 7a shows a simplified, two-dimensional visualization of the algorithm. A model consisting of three co-linear point is moved along a trajectory with three positions. At each position, a `FixedRangeSearch` is carried out around each point of the model. The figure shows a disadvantage of this approach: if the trajectory is not sampled densely enough, then some volumes along the path will not be checked for collisions as can be seen at the upper points in the graphic.

For a linear, non-parallel execution the time complexity of the algorithm is $O(MT \log n)$ where M is the number of points in the model, T is the number of sampled positions on the trajectory and n the number of points in the environment. For parallel execution, the time complexity is $O(\frac{MT}{p} \log n)$ where p is the number of worker processes. The complexity is as such because M times T searches in the k-d tree of the environment have to be done, where each search is of complexity $O(\log n)$. The complexity in the parallel case highlights that all M times T searches in the k-d tree can be carried out in parallel.

4.2. kd-CD

Instead of searching a fixed radius around every point of the model at each position on its trajectory like kd-CD-simple, this variant linearly connects the same point of the model at two consecutive positions on its trajectory and searches a fixed radius around all the line segments that are created in this manner.

Figure 7b shows a simplified, two-dimensional visualization of the algorithm. The model of three co-linear points is moved along a trajectory with three positions just as for the kd-CD-simple example. But instead of executing a `FixedRangeSearch` around each point of the model, a search is done around the line segments connecting the same point at two consecutive positions on the trajectory. The area that is searched this way is highlighted in orange and dark-green in the figure for the first and second search-pass, respectively.

This means that with T positions on the trajectory, this method will execute $M(T-1)$ k-d tree searches using `segmentSearch_all`. Thus, the time complexity of this algorithm is very similar to the one of kd-CD-simple $O(M(T-1) \log n)$ and becomes close to the one of kd-CD-simple for large numbers of T.

Since the trajectory can be less densely sampled than would be required for kd-CD-simple, kd-CD can thus require less search operations while maintaining a similar result quality. It also has the advantage that in contrast to the kd-CD-simple, the volumes of the environment that are searched for collisions are not spheres but cylinders with half spheres on both ends. This "smoothes" the found colliding points along the direction of movement of the model.

5. Depth of penetration calculation

Two variants to calculate depth of penetration will be presented: kd-PD-fast and kd-PD. They perform differently depending on the kind of input data and yield different results depending on the sampling rate of the model trajectory. kd-PD-fast is generally faster but produces only good results for objects protruding the path of the model through the environment. It does not produce correct results when the model moves alongside a wall and collides with it.

kd-PD-fast is an embarrassingly parallel operation just as the collision detection methods. The other variant, kd-PD, is easy to parallelize as well and the only part of kd-PD that has to be synchronized between workers is the updating of the penetration depth because it requires reading and checking the already stored depth of penetration per colliding point.

5.1. kd-PD-fast

This variant is a good heuristic for protruding sharp objects into the work space. At each position along the trajectory, it iterates through all points of the environment that are found to be colliding and finds the closest non-colliding point using `FindClosest`. The distance between the two points is then recorded as the depth of penetration. Thus, the time complexity of this algorithm is the same as for the collision detection algorithms and can be completely parallelized.

This variant works well for objects that "stick" into the path of the model because the penetration depth of the tip of that object will be about as deep as its distance to the closest non-colliding point. This method is shown to work well for automotive assembly lines as shown in prior work of the authors [1].

5.2. kd-PD

kd-PD represents a general penetration depth method. Consider figure 8 which illustrates this method. Figure 8 shows a top view of the train wagon model at one point of its trajectory inside the tunnel. It is shown colliding with the right hand side tunnel wall. The algorithm iterates over every point of the model P_n and finds its projection to the wagon center A_n. Since the central axis is the y-axis in the coordinate system of the train wagon (compare figure 2), this projection is simply done by setting the x and z coordinates to zero. Then a segment search using `segmentSearch_1NearestPoint` on the line segment from P_n to A_n

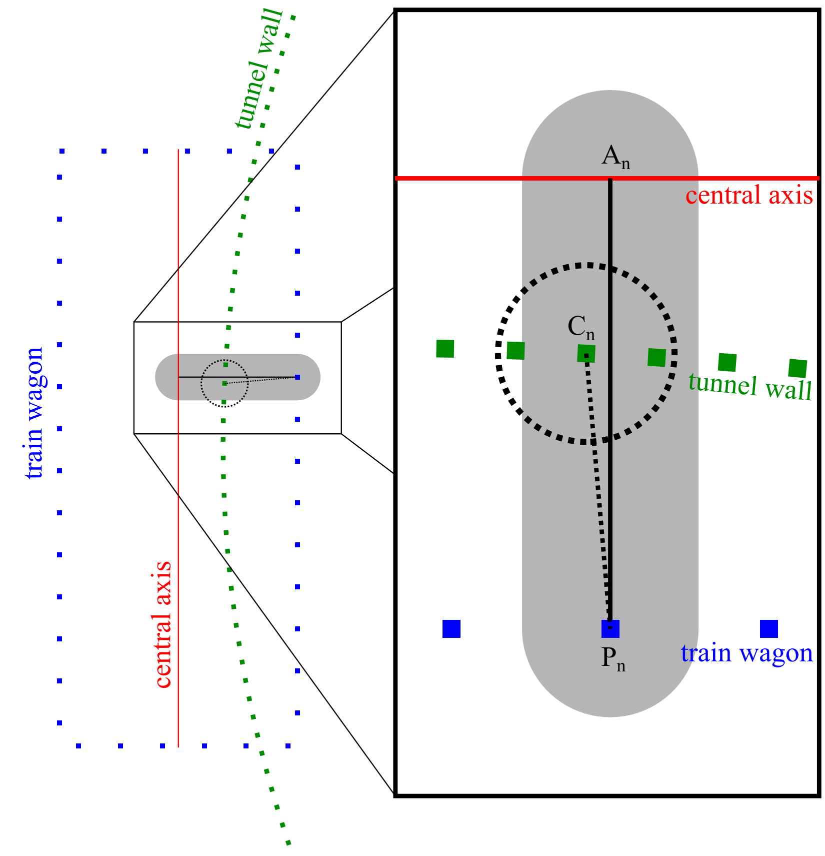


Figure 8: Left: a top view of the train wagon (blue) at a position through the tunnel (green). **Right:** a magnified and rotated part of the left figure with point names. The gray area represents the segment search volume between point P_n of the train wagon and point A_n on the wagon's central axis (red). The dotted black line is the distance between P_n and C_n which is the point that is found to be closest to P_n within the search area. The dotted circle shows the search radius around C_n. All points of the tunnel wall within this radius are updated with the same distance that C_n has to P_n if that distance is greater than the previously stored one.

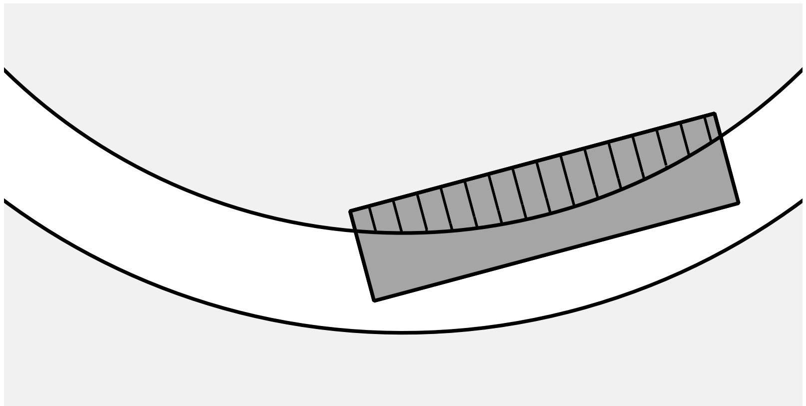


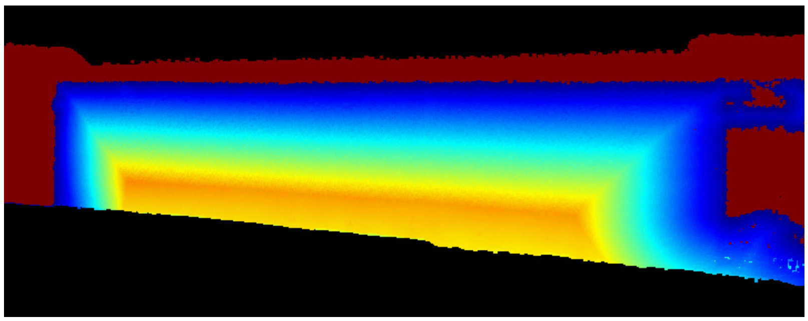
Figure 9: As the wagon (dark gray) moves along the tunnel (light gray), each point of the tunnel wall is updated with its maximum distance to the wagon exterior (stripes) on any point along the trajectory

is performed for every point of the model: for each point P_n the closest point C_n of the colliding environment within the search radius is found. A fixed range search using `FixedRangeSearch` of radius r around C_n is performed and all points within that search radius including C_n are collected. This collecting of points has to be performed because otherwise, many points of the environment are missed by `segmentSearch_1NearestPoint`. The distance between C_n and P_n is calculated and that distance is assigned to all points that are found by `FixedRangeSearch` if the new distance value is greater than the old one. This set of calculations is done for each point of the model on each position of its trajectory. In the end, every colliding point of the environment has attached to it the greatest distance found by this method over the whole trajectory. As is seen from figure 8, the maximum error of the calculated penetration distance is the size of the search radius.

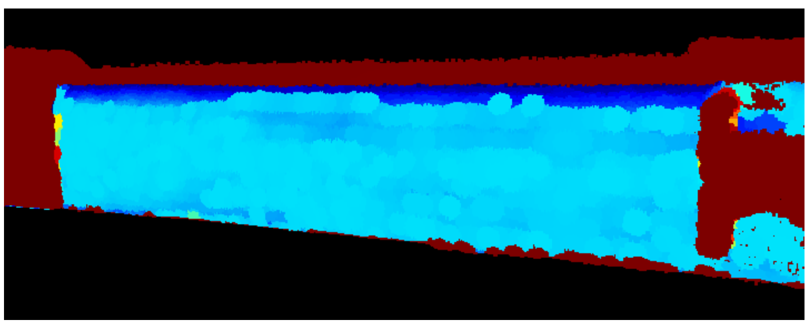
Figure 9 visualizes this method for two subsequent positions on the trajectory. The figure shows the calculated distances between each point of the model and each set of points in the colliding environment.

This method requires that the individual points of the trajectory are not further apart than the search radius. While this is also one of the reasons why this method is more computationally expensive than the first heuristic, it also yields better results when applied to a collision with the tunnel wall. Figure 10 illustrates the difference.

The time complexity in the non-linear case is the same as for kd-PD-fast and for the collision detection algorithms. In parallel execution, some time has to be



(a) Penetration depth as calculated by kd-PD-fast. The colors indicate the distance to the closest non-colliding point of the tunnel wall.



(b) Penetration depth as calculated by kd-PD. The colors indicate the maximum penetration depth of the tunnel wall into the moving train wagon on any point of its trajectory.

Figure 10: A comparison of the penetration depth as calculated by kd-PD-fast (top) and kd-PD (bottom). Both figures show a narrow piece of tunnel from the outside with the calculated penetration depth indicated by the point color. Non-colliding points are shown in dark red.

Figure 11: The Optech Lynx Mobile Mapper on the back of a train wagon.

Figure 12: A photo of the scanned train wagon with a bogie distance of 20 m.

spent synchronizing the access to the data structure that stores the currently closest penetration distance before updating it.

6. Experiments and results

A 3D point cloud of train tunnel was provided to us by the company TopScan GmbH. The point cloud contains 18.92 million points of outdoor data. The point cloud was collected by a Optech Lynx Mobile Mapper mounted on a van which was placed on a train wagon (see figure 11). TopScan also provided the trajectory data to us which is comprised of 23274 positions over a distance of 1144 m. The trajectory contains positional as well as orientation data.

Figure 13: The Riegl VZ-400 laser scanner set up next to the train wagon.

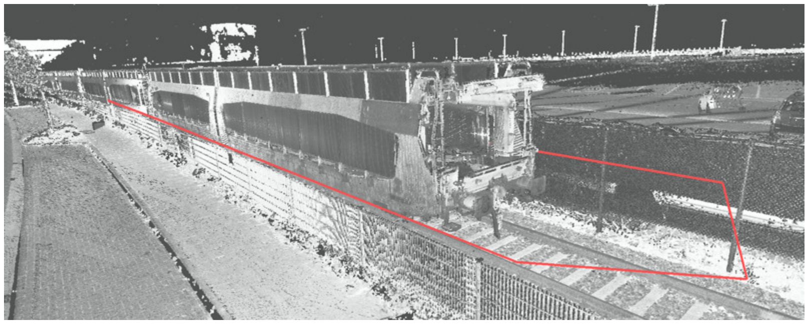


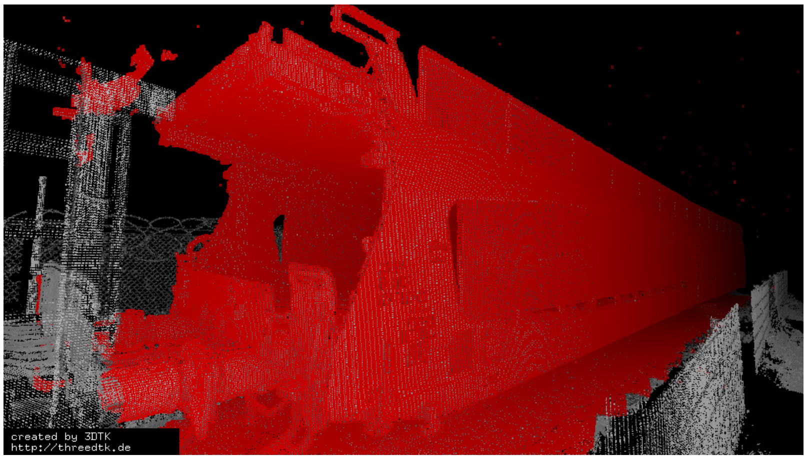
Figure 14: The registered point cloud of all seven scans. The red line connects the positions of the scanner.

To retrieve a point cloud of a suitable model to move through the environment, the train wagon that is seen in figure 12 was manually scanned using a RIEGL VZ-400 laser scanner (see figure 13). Seven scans were taken from all sides of the wagon and registered using 3DTK's SLAM implementation (figure 14).

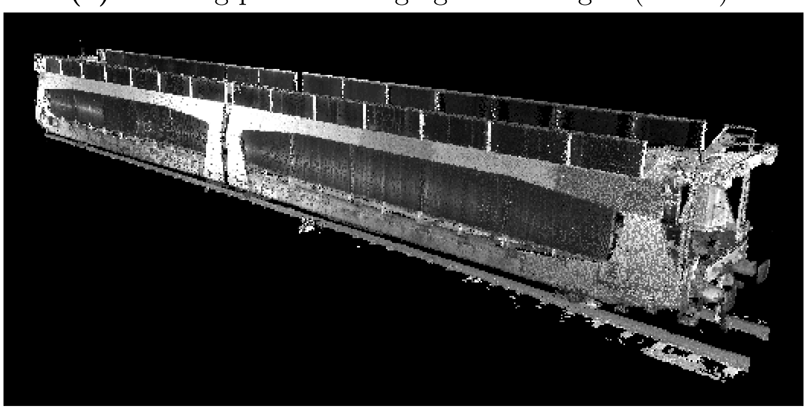
The train wagon is manually extracted from the resulting registered point cloud by using 3DTK's `show` application (see figure 15). As the train wheels are still part of the wagon, they will always result in an expected collision with the rails themselves.

It is then aligened inside the axis aligned bounding box of the wagon displayed in figure 2. The alignment process is shown in figure 16a. As calibration data of the precise location of the scanner relative to the environment is missing, our results can only serve a demonstration purpose of our methods (see figure 16b). The final point cloud of the wagon contained 2.5 million points.

The trajectory provided to the authors included orientation information in three degrees of freedom as well. Since a train wagon is mounted on two bogies and since the origin of the coordinate system of the train is located in its center (see figure 2), using this trajectory directly would mean that the wagon would rotate around its own center along the trajectory. This produces wrong results since instead, the bogies of the train have to remain on the tracks while the center follows accordingly. A new trajectory is calculated from the original trajectory by assuming a bogie distance of 20 m and moving the train wagon such that the center of both bogies is always on the original trajectory. Since this operation requires the original trajectory to be a continuous function and not a sampled trajectory, a spline is fitted across all points of the trajectory with a sum of squared residuals over all the spline's control points of 10 m. This amounts to the spline only a few millimeter away on average from the original trajectory. The FITPACK library [38] is used to calculate the spline. The

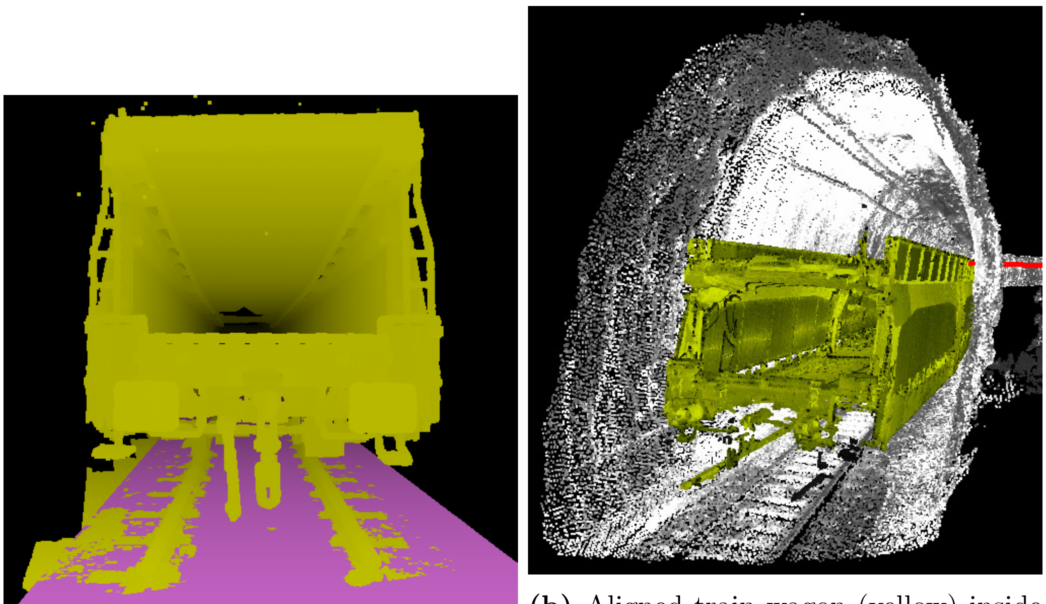


(a) Marking points belonging to the wagon (in red).



(b) The extracted model of the train wagon.

Figure 15: Extracting the point cloud of the train wagon.



(a) Frontal view of the train wagon and the rectangular base of its bounding box

(b) Aligned train wagon (yellow) inside the tunnel environment (gray) and trajectory (red).

Figure 16: Aligning the point cloud along the axis aligned bounding box of figure 2.

result of this computation also adjusted the yaw and pitch of the trajectory.

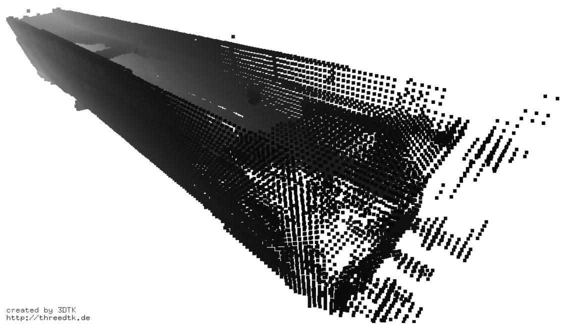
To benchmark the developed algorithms, the train wagon model as well as the trajectory are sampled with several different point distances. For the train wagon, the original amount of 2.5 million points is reduced using 3DTK's `scan_red` program which allows an octree based reduction of a point cloud with a given voxel size. As the search volume for collision detection must not contain any holes, a model of equidistant points is created by saving the center of each occupied octree voxel as point of the reduced model. This creates a 3D square lattice of points. Five different reductions of the train wagon point cloud are created to run benchmarks on them and are visualized in figure 17. Due to the structure of the underlying octree, the voxel size d_m is repeatedly halved starting from a maximum voxel size of 0.924 m and down to a voxel size of 5.8 cm. For each of the five reductions, the search radius is chosen to create a bounding sphere of an octree voxel of the respective size. That way, all space occupied by the model is searched for collisions without leaving any holes. This means that the voxel size d_m computes from the bounding sphere and search radius r as $d_m = \frac{2}{3}\sqrt{3}r$. Similarly, the trajectory is sampled such that the individual positions are between 5.8 cm and 14.78 m apart. Table 1 gives an overview of the chosen search radii, the according voxel size and trajectory position distances and the resulting number of points in the model and on the trajectory.

The benchmarks omit runtime results that only modify either the amount of points in the model or the amounts of positions in the trajectory. Both collision detections algorithms, kd-CD-simple and kd-CD, scale completely linearly and is completely parallelized by splitting the workload over different sets of points in the model or positions in the environment. The benchmarks are done on a Intel Core i5-4200U @ 1.6 GHz system with 16GB of system memory and only executed using a single thread.

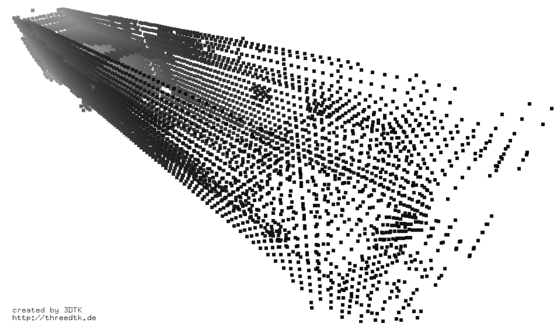
To test the claim that the structure gauge is an insufficient measure, given the provided environment and trajectory, a slice of the train wagon is moved through the tunnel. The slice is created by collapsing the y-coordinate of the train wagon model. The trajectory is created using above method but assuming a bogie length of zero. This effectively lets the slice travel exactly along the trajectory with the correct orientation perpendicular to the trajectory.

A video[1] was created to visually illustrate the difference between a structure gauge based method and kd-CD-simple. The video shows the train moving along its trajectory through the tunnel environment from the perspective of an observer who follows closely behind the train wagon. The view is split into three frames

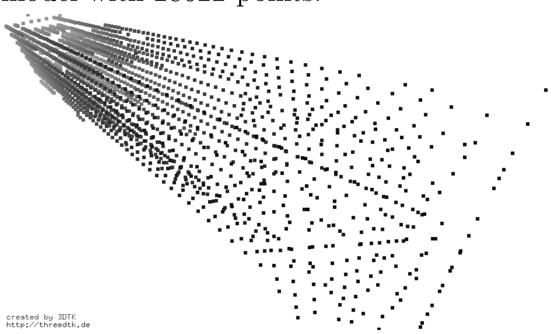
[1] `http://youtu.be/ylp4mD5XZaQ`



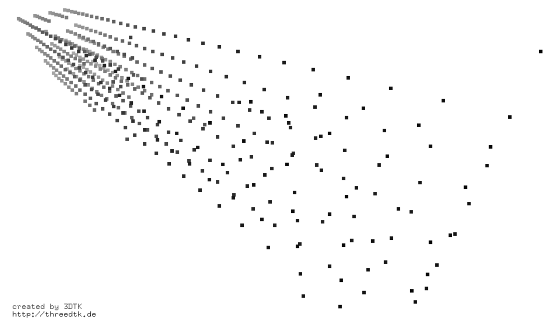
(a) Minimum point distance $d_m = 0.058\,\mathrm{m}$ for a search radius of $r = 0.05\,\mathrm{m}$ makes a model with 28622 points.



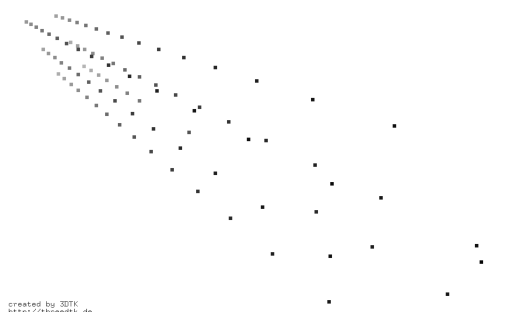
(b) Minimum point distance $d_m = 0.115\,\mathrm{m}$ for a search radius of $r = 0.1\,\mathrm{m}$ makes a model with 7546 points.



(c) Minimum point distance $d_m = 0.231\,\mathrm{m}$ for a search radius of $r = 0.2\,\mathrm{m}$ makes a model with 2041 points.



(d) Minimum point distance $d_m = 0.462\,\mathrm{m}$ for a search radius of $r = 0.4\,\mathrm{m}$ makes a model with 461 points.



(e) Minimum point distance $d_m = 0.924\,\mathrm{m}$ for a search radius of $r = 0.8\,\mathrm{m}$ makes a model with 93 points.

Figure 17: Five point models of the train wagon with different sampling densities. In all reductions, points are aligned in a 3D square lattice.

Table 1: The first column shows the choice of collision detection search radius r. The second column shows the resulting distance between the points of the wagon d_m and the points on the trajectory d_t. The third column shows the resulting number of points in the model. The fourth column shows the resulting number of points on the trajectory. The second and fourth column are extended as the results in figure 19 are calculated for higher distance values as well.

r in m	$d_m = d_t = \frac{2}{3}\sqrt{3}r$	#model	#trajectory
0.05	0.058	28622	19392
0.1	0.115	7546	9780
0.2	0.231	2041	4869
0.4	0.462	461	2434
0.8	0.924	93	1217
	1.848		609
	3.695		304
	7.390		152
	14.780		76

arranged next to each other. The leftmost frame shows the model of the train wagon in yellow moving through the environment in magenta. The center and right frame do not show the train wagon model for better visibility. The center frame shows the colliding points according to the structure gauge method in yellow. The rightmost frame shows the colliding points and their penetration depth as calculated by kd-CD-simple and kd-PD-fast. At multiple points during the video one observes that the center frame does not highlight points as colliding which are highlighted by the rightmost frames. Those points are most often found on the right tunnel wall as the train tracks make a turn to the right. This shows how the structure gauge based method is not able to find some of the collisions that are found by kd-CD-simple.

Figure 18 shows the influence of the search radius on the runtime of both collision detection variants, kd-CD-simple and kd-CD. While all other variables are kept constant, the algorithm is benchmarked with different search radii. The figure shows the runtime of both collision detection variants as well as the number of points that are found to be colliding in each variant. One can observe that the segment based variant finds more colliding points but that it is also slower than the fixed range search based method. Both variants increase exponentially in runtime with higher search radii. With small radii in the centimeter scale, which is desirable for precise results, the runtime of both variants stays below 10 seconds.

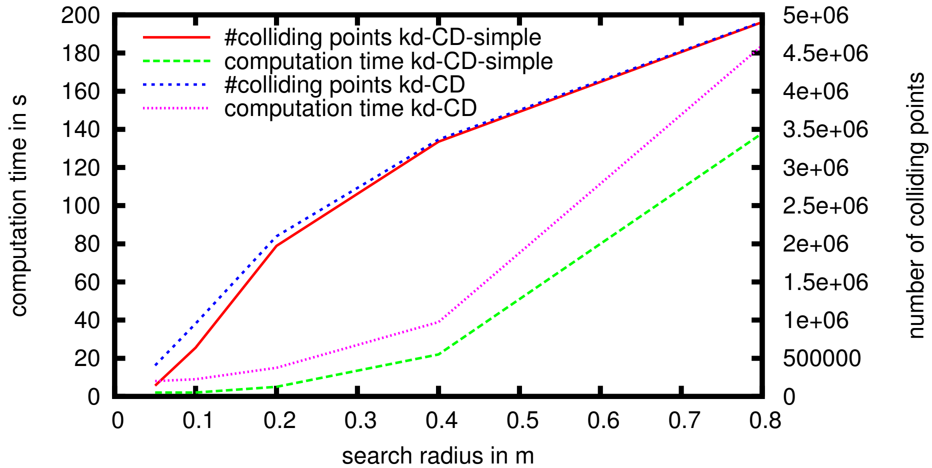


Figure 18: Computation time of both collision detection variants, kd-CD-simple and kd-CD, with different search radii r. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen to be $d_t = d_m = 0.231m$.

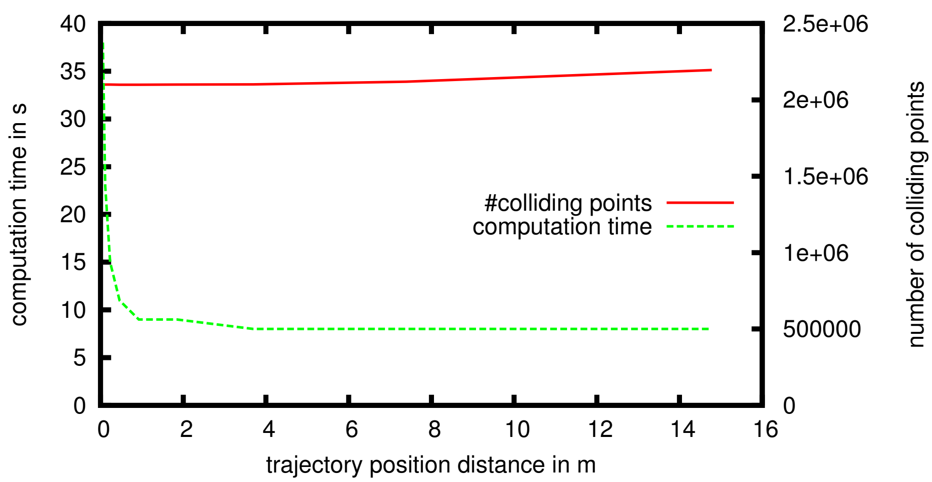


Figure 19: Computation time of kd-CD with different distances between individual points on the trajectory with a model sampled with $d_m = 0.231m$ and a search radius of $0.2m$.

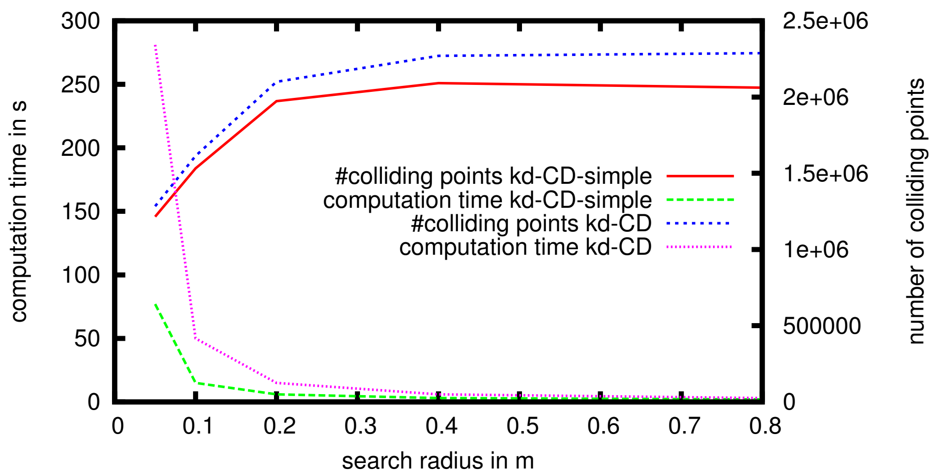


Figure 20: Computation time of both collision detection variants, kd-CD-simple and kd-CD, with different search radii r. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen such that $d_t = d_m = \frac{2}{3}\sqrt{3}r$.

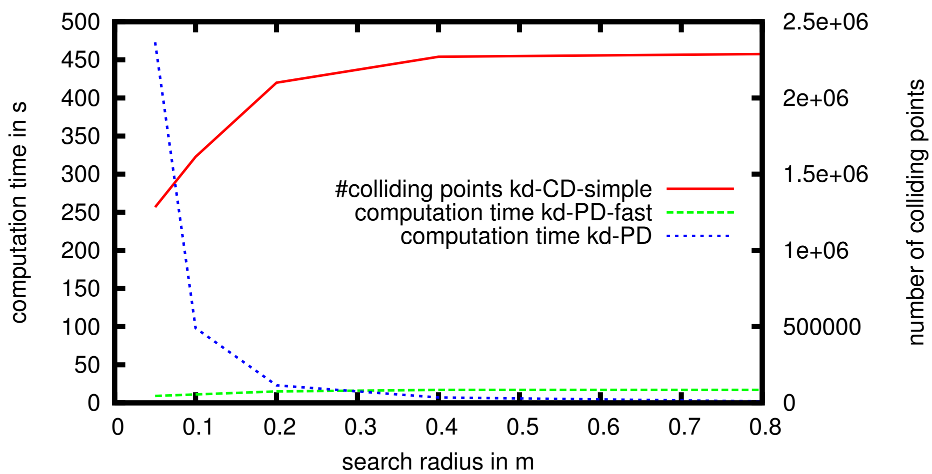


Figure 21: Computation time of both penetration depth variants, kd-PD-fast and kd-PD, with different search radii r. The distance between individual points on the trajectory d_t and the distance between points in the model d_m is chosen such that $d_t = d_m = \frac{2}{3}\sqrt{3}r$. Colliding points are computed using kd-CD.

In figure 19 the search radius is kept constant and the sampling rate of the trajectory is modified to investigate the dependency of the segment based collision detection method on the segment size. One can observe that as the segment size grows larger, the computation time quickly converges to a constant value of under 10 seconds. The amount of found colliding points slightly increases with larger segment sizes as more colliding points will be found inside the curvature of the tunnel wall.

Figure 20 shows a more realistic setup in the sense that not only the search radius is modified but also the sampling rate of the trajectory and train wagon model. If the search radius grows, lower sampling rates are possible because more volume is covered. For each value of search radius the sampling rates have been chosen such that no points of the environment are skipped as the model moves along its trajectory. The graph in figure 20 shows that the both algorithms, kd-CD-simple and kd-CD, quickly approaches runtimes below five seconds as the amount of required k-d tree searches decreases with higher search radii and thus lower sampling rates. On the other hand, the graph also shows, that with the lowest and thus most precise search radius of 5 cm which searches on a trajectory of $19,392$ positions a model of $28,622$ points, our k-d tree is able to make all required $19,392 \times 28,622 = 555,037,824$ k-d tree searches in only 77 s. This means that the average k-d tree search in a dataset of 18.92 Mill points takes 139 ns. This in turn means that collision detections of even complex models with up to 287000 points can be done in real time speed of 25.0 frames per second with the presented k-d search tree implementation.

In the last figure 21 the two depth of penetration methods, kd-PD-fast and kd-PD are compared. One can see that kd-PD-fast stays below 20 s of computation time. This is expected as the performance of kd-PD-fast only depends on the amount of colliding points found. We can observe that kd-PD-fast increases in runtime slightly as the mount of colliding points rises with increased search radius. kd-PD performs badly for very small search radii for which a large number of k-d tree searches have to be performed but quickly approaches runtime values below one minute as the search radius grows larger than 10 cm.

7. Conclusions and outlook

This paper presented a highly efficient k-d tree implementation which is used to perform collision detection of a sampled arbitrary point cloud against an environment of several million points. It is shown that even though this is a partly brute-force method as it checks all sampled points of the model, both, kd-CD-simple and kd-CD perform well enough such that real queries of densely sampled trajectories are completed in a matter of seconds. Two heuristics for calculating penetration depth, kd-PD-fast and kd-PD have been presented which work for different scenarios and have different precision and runtime properties.

For future work, several routes to improve these methods exist. More work has to be done to research which checks to abort the k-d tree traversal for different search geometries and input data perform best. Another easy way to increase the performance could be to change the sampling of the model from bounding spheres to different geometries like axis aligned bounding boxes which are similarly quick to check for collisions. Lastly, instead of checking every point of the model, a hierarchy of bounding spheres or other geometries could be used [11] but that would destroy the property of the current algorithm that the input model is allowed to arbitrarily deform.

Both variants, kd-CD-simple and kd-CD, are embarrassingly parallel operations. All k-d tree searches can be run in parallel and even updating of the boolean collision vector can be done in parallel as its values are only ever written but not read during collision detection. Thus, it should easily be possible to run the algorithm which is currently executed in series in parallel instead. Verifying the possible performance improvements of this measure is up to further research.

[1] J. Elseberg, D. Borrmann, J. Schauer, A. Nüchter, D. Koriath, U. Rautenberg, A sensor skid for precise 3d modeling of production lines, ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences II-5 (2014) 117–122. doi:10.5194/isprsannals-II-5-117-2014.
URL http://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/II-5/117/2014/

[2] J. Siegmann, Lichtraumprofil und Fahrzeugbegrenzung im europäischen Schienenverkehr, http://www.forschungsinformationssystem.de/servlet/is/325031/, [Online; accessed 2014-07-14] (2011).

[3] O. Lueger, Krümmungsverhältnisse, in: Lexikon der gesamten Technik und ihrer Hilfswissenschaften, Stuttgart / Leipzig: DVA, 1904, pp. 718–724.

[4] EBO, Eisenbahn-Bau- und Betriebsordnung, http://www.gesetze-im-internet.de/ebo/anlage_1_67.html, [Online; accessed 2014-07-14] (1967).

[5] A. Nüchter, J. Elseberg, P. Schneider, D. Paulus, Study of parameterizations for the rigid body transformations of the scan registration problem, Computer Vision and Image Understanding 114 (8) (2010) 963 – 980. doi:http://dx.doi.org/10.1016/j.cviu.2010.03.007.

URL http://www.sciencedirect.com/science/article/pii/S107731421000072X

[6] P. Jiménez, F. Thomas, C. Torras, 3d collision detection: a survey, Computers & Graphics 25 (2) (2001) 269–285.

[7] M. Lin, S. Gottschalk, Collision detection between geometric models: A survey, in: Proc. of IMA conference on mathematics of surfaces, Vol. 1, 1998, pp. 602–608.

[8] J. Bender, K. Erleben, J. Trinkle, Interactive simulation of rigid body dynamics in computer graphics, in: Computer Graphics Forum, Vol. 33, Wiley Online Library, 2014, pp. 246–270.

[9] D. Mainzer, G. Zachmann, Collision detection based on fuzzy scene subdivision, in: Symposium on GPU Computing and Applications (Singapore, 2013), Vol. 3, 2014.

[10] M. Tang, D. Manocha, J. Lin, R. Tong, Collision-streams: fast gpu-based collision detection for deformable models, in: Symposium on interactive 3D graphics and games, ACM, 2011, pp. 63–70.

[11] C. Tzafestas, P. Coiffet, Real-time collision detection using spherical octrees: virtual reality application, in: Robot and Human Communication, 1996., 5th IEEE International Workshop on, 1996, pp. 500–506. doi:10.1109/ROMAN.1996.568888.

[12] J. S. Muñoz, C. A. D. León, H. T. Gómez, Development of a hierarchy collision detection algorithm in order to implement a laparoscopic surgical simulator, Revista QUID (19).

[13] J. Hummel, R. Wolff, T. Stein, A. Gerndt, T. Kuhlen, An evaluation of open source physics engines for use in virtual reality assembly simulations, in: Advances in Visual Computing, Springer, 2012, pp. 346–357.

[14] J. Klein, G. Zachmann, Point cloud collision detection, in: Computer Graphics Forum, Vol. 23, Wiley Online Library, 2004, pp. 567–576.

[15] A. Hermann, F. Drews, J. Bauer, S. Klemm, A. Roennau, R. Dillmann, Unified gpu voxel collision detection for mobile manipulation planning, in: Intelligent Robots and Systems (IROS), 2014, 2014.

[16] D. Jung, K. K. Gupta, Octree-based hierarchical distance maps for collision detection, in: Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on, Vol. 1, IEEE, 1996, pp. 454–459.

[17] W. Fan, B. Wang, J.-C. Paul, J. Sun, An octree-based proxy for collision detection in large-scale particle systems, Science China Information Sciences 56 (1) (2013) 1–10.

[18] H. Y. Wang, S. G. Liu, A collision detection algorithm using aabb and octree space division, in: Advanced Materials Research, Vol. 989, Trans Tech Publ, 2014, pp. 2389–2392.

[19] S. Ar, B. Chazelle, A. Tal, Self-customized bsp trees for collision detection, Computational Geometry 15 (1) (2000) 91–102.

[20] M. Held, J. T. Klosowski, J. S. Mitchell, Evaluation of collision detection methods for virtual reality fly-throughs, in: Canadian Conference on Computational Geometry, Citeseer, 1995, pp. 205–210.

[21] A. Garcia-Alonso, N. Serrano, J. Flaquer, Solving the collision detection problem, Computer Graphics and Applications, IEEE 14 (3) (1994) 36–43.

[22] D. Faas, J. M. Vance, Brep identification during voxel-based collision detection for haptic manual assembly, in: ASME 2011 World Conference on Innovative Virtual Reality, American Society of Mechanical Engineers, 2011, pp. 145–153.

[23] P. M. Hubbard, Approximating polyhedra with spheres for time-critical collision detection, ACM Transactions on Graphics (TOG) 15 (3) (1996) 179–210.

[24] H. A. Sulaiman, M. A. Othman, M. M. Ismail, M. Said, M. Alice, A. Ramlee, M. H. Misran, A. Bade, M. H. Abdullah, Distance computation using axis aligned bounding box (aabb) parallel distribution of dynamic origin point, in: Emerging Research Areas and 2013 International Conference on Microelectronics, Communications and Renewable Energy (AICERA/ICMiCR), 2013 Annual International Conference on, IEEE, 2013, pp. 1–6.

[25] S. Gottschalk, M. C. Lin, D. Manocha, Obbtree: A hierarchical structure for rapid interference detection, in: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, 1996, pp. 171–180.

[26] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, K. Zikan, Efficient collision detection using bounding volume hierarchies of k-dops, Visualization and Computer Graphics, IEEE Transactions on 4 (1) (1998) 21–36.

[27] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, M. Gross, Optimized spatial hashing for collision detection of deformable objects, Tech. rep., Technical report, Computer Graphics Laboratory, ETH Zurich, Switzerland (2003).

[28] G. v. d. Bergen, Efficient collision detection of complex deformable models using aabb trees, Journal of Graphics Tools 2 (4) (1997) 1–13.

[29] J. D. Cohen, M. C. Lin, D. Manocha, M. Ponamgi, I-collide: An interactive and exact collision detection system for large-scale environments, in: Proceedings of the 1995 symposium on Interactive 3D graphics, ACM, 1995, pp. 189–ff.

[30] R. G. Luque, J. L. Comba, C. M. Freitas, Broad-phase collision detection using semi-adjusting bsp-trees, in: Proceedings of the 2005 symposium on Interactive 3D graphics and games, ACM, 2005, pp. 179–186.

[31] T. Sellis, N. Roussopoulos, C. Faloutsos, The r+-tree: A dynamic index for multi-dimensional objects.

[32] J. Elseberg, S. Magnenat, R. Siegwart, A. Nüchter, Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration, Journal of Software Engineering for Robotics 3 (1) (2012) 2–12.

[33] D. M. Mount, S. Arya, Ann: a library for approximate nearest neighbor searching, 2005.
URL `http://www.cs.umd.edu/~mount/ANN/`

[34] S. Magnenat, libnabo.
URL `https://github.com/ethz-asl/libnabo`

[35] M. Muja, D. G. Lowe, Flann - fast library for approximate nearest neighbors.
URL `http://www.cs.ubc.ca/research/flann/`

[36] A. Nuchter, H. Surmann, K. Lingemann, J. Hertzberg, S. Thrun, 6d slam with an application in autonomous mine mapping, in: Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on, Vol. 2, IEEE, 2004, pp. 1998–2003.

[37] T. Larsson, T. Akenine-Möller, E. Lengyel, On faster sphere-box overlap testing, journal of graphics, gpu, and game tools 12 (1) (2007) 3–8.

[38] P. Dierckx, Curve and surface fitting with splines, Oxford University Press, Inc., 1993.