# 9. Containment in Convex Polytopes using $k$-D trees.

Several important features of Mastodon rely on the fast retrieval of data items (spots and links) close to specific 3D positions. For instance we need to do so when you move the mouse close to the drawing of a spot in a view, to retrieve the spot in question. Or to determine what spots must to be painted in a BDV view, depending on the zoom level, position, rotation and field of view. There exists several well established algorithms and techniques to do that, in the case where the bounding-box in which we need to retrieve data items is a 3D rectangle aligned with the X, Y, Z axes of the dataset.

In our case it is not. In BDV you can rotate the view around an arbitrary angle. The XY view plane does not match an orthogonal plane of the dataset. Also the planes that make the bounds of the field of view are not aligned with these axes. In our case, the field of view is a 'Convex Polytope'. It is a portion of 3D space delimited by a set of planes[11]. Think of an ideal diamond. Each facet of the diamond would be one of the planes. The interior of the diamond would be the convex polytope. Our goal is to know what are the points that are inside this volume so that we can *e.g.* paint them without losing time painting the ones not in the field of view.

At the time of the development of Mastodon, there was no published algorithm for the fast retrieval of points in a convex polytope. Tobias derived such an algorithm in 2016, and it is detailed in this chapter. To the best of our knowledge this is unpublished.

## 9.1. Introduction.

The problem we want to solve is the following: Given a set of points $\mathbf{x} \in \mathbb{R}^k$ and a convex polytope in $\mathbb{R}^k$, partition the set of points into those inside and outside the polytope.

We assume that the points are stored in a $k$-D tree (formally defined below). We start by deriving an algorithm that, given a hyperplane, partitions the set of points into points in the positive and negative half-space of the hyperplane, respectively. We then give an algorithm that, given a convex polytope (a set of hyperplanes), partitions the set of points into points that are inside and outside the polytope, respectively.

## 9.2. $k$-D trees.

We assume that the points are stored in a $k$-D tree which can be defined as follows.

**Definition 1 (binary point tree)** *We define the set of binary trees with points $\mathbf{x} \in \mathbb{R}^k$ stored in the nodes as*

$$\mathcal{T}_k = \{\perp\} \cup \left\{(\mathbf{x}, s, L, R) \mid \mathbf{x} \in \mathbb{R}^k,\ 1 \leq s \leq k,\ L, R \in \mathcal{T}_k\right\}$$

*where $\perp$ denotes the empty tree and $s$ is called the splitting dimension.*

[11] https://en.wikipedia.org/wiki/Convex_polytope

**Definition 2 (min and max coordinate of a tree)** *Let $T \in \mathcal{T}_k$ and $1 \le s \le k$. We define the min coordinate in dimension d as*

$$\min_d(T) = \begin{cases} +\infty & \text{if } T = \perp \\ \min\{x_d, \min_d(L), \min_d(R)\} & \text{if } T = (\mathbf{x}, s, L, R). \end{cases}$$

*We define the max coordinate in dimension d as*

$$\max_d(T) = \begin{cases} -\infty & \text{if } T = \perp \\ \max\{x_d, \max_d(L), \max_d(R)\} & \text{if } T = (\mathbf{x}, s, L, R), \end{cases}$$

*where $x_i$ denotes the ith component of vector $\mathbf{x}$.*

**Definition 3 ($k$-D tree)** *We define the set of k-D trees as*

$$\mathcal{T}_{kD} = \{\perp\} \;\cup\; \{(\mathbf{x}, s, L, R) \in \mathcal{T}_k \mid \max_s(L) \le \mathbf{x}_s \le \min_s(R),\; L, R \in \mathcal{T}_{kD}\}.$$

## 9.3. Sub-tree bounding boxes.

The algorithms presented in the following are all based on recursively visiting the nodes of a $k$-D tree in a depth-first search. While doing this, we maintain the bounding box of all coordinates in the sub-tree rooted in the visited node. The recursion is given in Algorithm 1. We use $\mathbf{x}[i \mapsto y]$ to denote the vector $\mathbf{x}$ with the $i$th component replaced by $y$.

**Algorithm 1:** Sub-tree bounding boxes.

```
Procedure visit((x, s, L, R), x^min, x^max):
    if L ≠ ⊥ then
        visit(L, x^min, x^max[s ↦ x_s])
    if R ≠ ⊥ then
        visit(R, x^min[s ↦ x_s], x^max)
```

It is easy to show that $\forall d, 1 \le d \le k \;:\; \max_d(T) \le x_d^{\max} \wedge \min_d(T) \ge x_d^{\min}$ is an invariant of the recursion in *visit* $\left(T \in \mathcal{T}_{kD}, \mathbf{x}^{\min}, \mathbf{x}^{\max}[s \mapsto x_s]\right)$.

## 9.4. Splitting $k$-D tree by a hyperplane.

Let $P = (\mathbf{n}, m)$ with $\mathbf{n} \in \mathbb{R}^k, m \in \mathbb{R}$ denote a $k$-dimensional hyperplane. Point $\mathbf{x} \in \mathbb{R}^k$ is *on* the plane iff $\mathbf{x} \cdot \mathbf{n} = m$; it is *above* the plane iff $\mathbf{x} \cdot \mathbf{n} \ge m$; it is *below* the plane iff $\mathbf{x} \cdot \mathbf{n} < m$.

Consider a set $X$ of points $\mathbf{x} \in \mathbb{R}^k$. Let a bounding box of $X$ be given by $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}\right)$ such that

$$\forall \mathbf{x} \in X,\; \forall d, 1 \le d \le k \;:\; x_d^{\min} \le x_d \le x_d^{\max}.$$

To determine whether all points in $X$ lie above or below a hyperplane $(\mathbf{n}, m)$ respectively, it is sufficient to check the bounding box corner that is furthest along the negative or positive direction of the normal $\mathbf{n}$. This is formalized in functions *allAbove* and *allBelow* in Algorithm 2.

---

**Algorithm 2:** Bounding box above or below plane.

---

**Function** *allAbove* $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, (\mathbf{n}, m)\right) \ : \ b \in \mathbb{B}$

$$\mathbf{x} := (x_1, \ldots, x_n)\,,\ x_d = \begin{cases} x_d^{\min} & \text{if } n_d \geq 0 \\ x_d^{\max} & \text{if } n_d < 0 \end{cases}$$

**return** $\mathbf{x} \cdot \mathbf{n} \geq m$

**Function** *allBelow* $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, (\mathbf{n}, m)\right) \ : \ b \in \mathbb{B}$

$$\mathbf{x} := (x_1, \ldots, x_n)\,,\ x_d = \begin{cases} x_d^{\min} & \text{if } n_d < 0 \\ x_d^{\max} & \text{if } n_d \geq 0 \end{cases}$$

**return** $\mathbf{x} \cdot \mathbf{n} < m$

---

It is easy to show that

- if *allAbove* $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, (\mathbf{n}, m)\right) = $ *true* then all points in the bounding box $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}\right)$ are above the plane, and
- if *allBelow* $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, (\mathbf{n}, m)\right) = $ *true* then all points in the bounding box $\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}\right)$ are below the plane.

Given these functions we can devise an algorithm that partitions points in a $k$-D tree $T = (\mathbf{x}, s, L, R) \in \mathcal{T}_{kD}$ into sets $A$ (points above the hyperplane) and $B$ (points below the hyperplane) as follows: Check whether $\mathbf{x}$ is above or below the hyperplane and add it to $A$ or $B$ accordingly. Determine bounding boxes for $L$ and $R$ as in Algorithm 1 and test whether these are *allAbove* or *allBelow* the hyperplane. If so, add all points in sub-trees $L$ and $R$ to $A$ or $B$, respectively. Otherwise recursively descent into $L$ and $R$.

This computation can be made more efficient by eliminating certain checks for $L$ and $R$. For example, assume that $\mathbf{x}$ is *above* the hyperplane. Further assume that $n_s \geq 0$. Because we recursively descended into $T$, we already know that the bounding box of $T$ is not *allAbove* the hyperplane. This means that the bounding box corner furthest to the negative normal direction is not *above* the hyperplane. Now, the bounding box for $L$ only differs from the bounding box of $T$ in that $x_s^{\max} = x_s$. Because $n_s \geq 0$, the bounding box corner of $L$ furthest to the negative normal direction will have $s$th component equal to $x_s^{\min}$. This means that the bounding box corner furthest to the negative normal direction for $L$ has the same coordinates as that for $T$. Therefore, we already know that $L$ is not *allAbove* the hyperplane. Consequently we can eliminate the *aboveAll* check for $L$ and recursively descent immediately. Similar considerations can be made for other combinations of sign of $n_s$ and $L$ or $R$.

The resulting algorithm is given in Algorithm 3, where we use *all* $(T)$ to denote the set of all points in the sub-tree $T$.

**Algorithm 3:** Split $k$-D tree points on hyperplane. Given a $k$-D tree and a hyperplane, the function *split* computes a partition of the points in the tree into sets $A$ and $B$ of point *above* and *below* the hyperplane, respectively.

**Function** $split\left((\mathbf{x}, s, L, R), \mathbf{x}^{\min}, \mathbf{x}^{\max}, (\mathbf{n}, m)\right) \;:\; (A, B) \in \mathcal{P}\left(\mathbb{R}^k\right) \times \mathcal{P}\left(\mathbb{R}^k\right)$

$p := \mathbf{x} \cdot \mathbf{n} \geq m$ // set $p$ if $\mathbf{x}$ is above hyperplane
$q^L := n_s < 0$ // set $q^L$ if $\mathbf{n}$ points towards left child
$q^R := n_s \geq 0$ // set $q^R$ if $\mathbf{n}$ points towards right child

// handle $\mathbf{x}$
**if** $p$ **then**
  $(A, B) := (\{\mathbf{x}\}, \varnothing)$
**else**
  $(A, B) := (\varnothing, \{\mathbf{x}\})$

// handle left child
$(A, B) := (A, B) \cup splitSubtree\left(L, \mathbf{x}^{\min}, \mathbf{x}^{\max}[s \mapsto x_s], (\mathbf{n}, m), p, q^L\right)$

// handle right child
$(A, B) := (A, B) \cup splitSubtree\left(R, \mathbf{x}^{\min}[s \mapsto x_s], \mathbf{x}^{\max}, (\mathbf{n}, m), p, q^R\right)$

**return** $(A, B)$

**Function** $splitSubtree\left(T, \mathbf{x}^{\min}, \mathbf{x}^{\max}, P, p, q\right) \;:\; (A, B) \in \mathcal{P}\left(\mathbb{R}^k\right) \times \mathcal{P}\left(\mathbb{R}^k\right)$
**if** $p \wedge q \wedge allAbove\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, P\right)$ **then**
  **return** $(all(T), \varnothing)$
**else if** $\neg p \wedge \neg q \wedge allBelow\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, P\right)$ **then**
  **return** $(\varnothing, all(T))$
**else**
  **return** $split\left(T, \mathbf{x}^{\min}, \mathbf{x}^{\max}, P\right)$

## 9.5. Splitting $k$-D tree into Inside and Outside of a Convex Polytope.

Now assume that we are given a convex polytope $C = \{P_1, \ldots P_h\}$ defined by hyperplanes $P_i = \left(\mathbf{n}^i, m^i\right)$ such that points $\mathbf{x} \in \mathbb{R}^k$ are *inside* $C$ if they are above all hyperplanes $P_i$ and *outside* $C$ otherwise. We want to partition points in a $k$-D tree $T = (\mathbf{x}, s, L, R) \in \mathcal{T}_{kD}$ into sets $A$ and $B$ of points inside and outside the polytope, respectively.

Using the same reasoning as in Section 9.4 we can devise an algorithm that partitions points in a $k$-D tree $T = (\mathbf{x}, s, L, R) \in \mathcal{T}_{kD}$ into sets $A$ (points inside the polytope) and $B$ (points outside the polytope) as follows: Check whether $\mathbf{x}$ is above all hyperplanes $P_i$. If so, add $\mathbf{x}$ to $A$, otherwise add it to $B$. Determine bounding boxes for $L$ and $R$ as in Algorithm 1, and test whether these are *allAbove* and *allBelow* all hyperplanes $P_i$. If the bounding box for $L$ (or $R$) is above *all* of the hyperplanes $P_i$, add all points in the sub-tree $L$ (or $R$) to set $A$. If the bounding box for $L$ (or $R$) is below *a single one* of the hyperplanes $P_i$, add all points in the sub-tree $L$ (or $R$) to set $B$. Otherwise recursively descent into $L$ and $R$.

We can make the following considerations to make the computation more efficient:

- Certain checks for individual hyperplanes can be eliminated by the same reasoning as in Section 9.4.
- If a sub-tree is *allBelow* a single hyperplane, we can stop checking further hyperplanes. The recursion can be terminated and the whole sub-tree can be added to the *outside* set $B$.
- If a sub-tree is *allAbove* a given hyperplane, all sub-trees further down the recursion will be *allAbove* this hyperplane as well. Consequently, that hyperplane can be removed from the set of hyperplanes to consider for this branch of the recursion. If in this process the set of hyperplanes becomes empty, recursion can be terminated and the whole sub-tree can be added to the *inside* set $A$.

The resulting algorithm is given in Algorithm 4, where $P_i = \left(\mathbf{n}^i, m^i\right)$ and we use *all* $(T)$ to denote the set of all points in the sub-tree $T$.

**Algorithm 4:** Partition $k$-D tree points into interior and exterior of a polytope. Given a $k$-D tree and a convex polytope, the function *clip* computes a partition of the points in the tree into sets $A$ and $B$ of point *inside* and *outside* the polytope, respectively.

**Function** $clip\left((\mathbf{x}, s, L, R), \mathbf{x}^{\min}, \mathbf{x}^{\max}, \{P_1, \dots, P_h\}\right) \ : \ (A, B) \in \mathcal{P}\left(\mathbb{R}^k\right) \times \mathcal{P}\left(\mathbb{R}^k\right)$

- **foreach** $1 \le i \le h$ **do**
  - $p_i := \mathbf{x} \cdot \mathbf{n}^i \ge m^i$ `// set` $p_i$ `if` $\mathbf{x}$ `is above hyperplane` $P_i$
  - $q_i^L := n_s^i < 0$ `// set` $q_i^L$ `if` $\mathbf{n}^i$ `points towards left child`
  - $q_i^R := n_s^i \ge 0$ `// set` $q_i^R$ `if` $\mathbf{n}^i$ `points towards right child`
- $\mathbf{p} := (p_1, \dots, p_h)$
- $\mathbf{q}^L := \left(q_1^L, \dots, q_h^L\right)$
- $\mathbf{q}^R := \left(q_1^R, \dots, q_h^R\right)$
- `// handle` $\mathbf{x}$
- **if** $\bigwedge_i p_i$ **then**
  - $(A, B) := (\{\mathbf{x}\}, \varnothing)$
- **else**
  - $(A, B) := (\varnothing, \{\mathbf{x}\})$
- `// handle left child`
- $(A, B) := (A, B) \cup$
  $clipSubtree\left(L, \mathbf{x}^{\min}, \mathbf{x}^{\max}[s \mapsto x_s], \mathbf{p}, \mathbf{q}^L, \{P_1, \dots, P_h\}\right)$
- `// handle right child`
- $(A, B) := (A, B) \cup$
  $clipSubtree\left(R, \mathbf{x}^{\min}[s \mapsto x_s], \mathbf{x}^{\max}, \mathbf{p}, \mathbf{q}^R, \{P_1, \dots, P_h\}\right)$
- **return** $(A, B)$

**Function** $clipSubtree\left(T, \mathbf{x}^{\min}, \mathbf{x}^{\max}, \mathbf{p}, \mathbf{q}, \{P_1, \dots, P_h\}\right) \ : \ (A, B) \in \mathcal{P}\left(\mathbb{R}^k\right) \times \mathcal{P}\left(\mathbb{R}^k\right)$

- $\mathcal{P} := \{P_1, \dots, P_h\}$
- **foreach** $1 \le i \le h$ **do**
  - **if** $p_i \wedge q_i \wedge allAbove\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, P_i\right)$ **then**
    - $\mathcal{P} := \mathcal{P} \setminus \{P_i\}$
  - **else if** $\neg p_i \wedge \neg q_i \wedge allBelow\left(\mathbf{x}^{\min}, \mathbf{x}^{\max}, P_i\right)$ **then**
    - **return** $(\varnothing, all(T))$
- **if** $\mathcal{P} = \varnothing$ **then**
  - **return** $(all(T), \varnothing)$
- **else**
  - **return** $clip\left(T, \mathbf{x}^{\min}, \mathbf{x}^{\max}, \mathcal{P}\right)$

### 9.6. Source code availability.

Implementations of the algorithms discussed above are provided in ImgLib2 [10]. The *split* algorithm for splitting a $k$-D tree by a hyperplane is implemented in `SplitHyperPlaneKDTree`[12] in the `kdtree` package. The *clip* algorithm for splitting a $k$-D tree into inside and outside of a convex polytope is implemented in `ClipConvexPolytopeKDTree`[13] in the same package.

[12] net.imglib2.algorithm.kdtree.SplitHyperPlaneKDTree
[13] net.imglib2.algorithm.kdtree.ClipConvexPolytopeKDTree