An introduction to computational geometry

Tim Prokosch[1], based on the slides provided by Prof. Dr. Heike Leitte

Winter term 2022/23 at the RPTU Kaiserslautern

[1]Suggestions and corrections are welcome and can be sent to prokosch@rhrk.uni-kl.de

Contents

List of Algorithms

Chapter 1

Introduction

1.1 General information

The area of *Computational Geometry* was established between 1975 and 1980. Its goals were the design and analysis of algorithms and data structures for geometric problems. Which are problems, often motivated by applications in the real world that can be formulated in terms of geometric objects, such as points, lines, polygons and similar. Especially the design of efficient algorithms for these problems is a major goal of computational geometry. In the following we will list some popular applications of computational geometry:

- ▷ *Computer graphics*, e.g. for rendering scenes in video games.
- ▷ *Data visualization*, e.g. for charts and maps.
- ▷ *Geographic information systems*, e.g. for routing in navigation systems or for finding the nearest warehouse to a customer.
- ▷ *Computer vision* and *pattern recognition*, e.g. for object detection in images by artificial intelligence.
- ▷ *Robotics*, e.g. for autonomous driving and collision avoidance.

1.2 Motivating problems

To further motivate the field of computational geometry, we will list some problems that can be solved using algorithms developed in the context of computational geometry.

1. **Problem:** Finding the nearest warehouse to a customer.
   Solution: Use *Voronoi diagrams*.
   Approach: Fortune's algorithm.

2. **Problem:** Find places where roads cross rivers on a map.
   Solution: Use knowledge about *line segment intersection* problems.
   Approach: Bentley-Ottmann (sweep line) algorithm.

3. **Problem:** Given a long-lat coordinate, return the country it is in.
   Solution: Use knowledge about *point location* problems.
   Approach: Trapezoidal maps.

4. **Problem:** Find datapoints that lie in a given range.
   Solution: Use *range trees*.
   Approach: kD-Trees.

5. **Problem:** Place cameras, such that a room is fully monitored.
   Solution: Use *polygon triangulation* and a *3-coloring*.
   Approach: Half-edge-data structure.

1.3 Geometric objects

As mentioned above, computational geometry is concerned with geometric objects. As we will mainly focus on 2D-problems, we will only consider the most relevant 2D-objects.

Definition 1.1 (Geometric objects)

In the following we will define a...

- ▷ *point* as the element of d-dimensional euclidean space $\mathbb{R}^d$,
- ▷ *line* as set of the form $L_{pq} := \{\lambda p + (1-\lambda)q : \lambda \in \mathbb{R}\}$ for two points p and q,
- ▷ *line segment* as set of the form $\overline{pq} := \{\lambda p + (1-\lambda)q : \lambda \in [0,1]\}$ for two points p and q,
- ▷ *polygon chain* as the tuple $P := (\overline{p_i p_{i+1}})_{i\in\{0,\dots,n-1\}}$ with points $p_0, \dots, p_n$. Moreover, P is called *simple*, if it does not intersect itself and
- ▷ *polygon* as he set P of all points that are bounded by a simple polygon chain $(\overline{p_i p_{i+1}})_{i\in\{0,\dots,n-1\}}$ with $p_0 = p_n$.

1.4 Runtime analysis

Since we are interested in the efficiency of the presented algorithms, we will briefly introduce central concepts for the runtime analysis of algorithms. First, we have to agree on some primitive operations that can be executed in constant time. The most important primitive operations are:

- ▷ Calculating $g \cap h$ for two lines g and h.
- ▷ Calculating $g \cap h$ for two line segments g and h.
- ▷ Checking $p \in s$ for a point p and a line segment s.
- ▷ Checking $p \in H$ for a point p and a half-space H.

To get a better intuition of operations that are primitive, we will list some that are not. For example:

- ▷ Checking $p \in P$ for a polygon P.
- ▷ Calculating $P \cap Q$ for two polygons P and Q.
- ▷ Calculating $\bigcap_{i\in I} g_i$ for an arbitrary finite family of line segments g_i.

To make the runtime analysis of algorithms more precise, we will introduce the following notation. Albeit very mathematical, it will proof itself to be of great use in the following sections.

Definition 1.2 ($\mathcal{O}$-calculus)

The *asymptotic runtime* of a function f with respect to another function g can be classified as follows[1]:

- ▷ $f \in \Omega(g) := \{h : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R}_+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : g(n) \leq cg(n)\}$

▷ $f \in \Theta(g) := \{h : \mathbb{N} \to \mathbb{R}_+ : \exists c_1, c_2 \in \mathbb{R}_+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : c_1 g(n) \leq h(n) \leq c_2 g(n)\}$

▷ $f \in \mathcal{O}(g) := \{h : \mathbb{N} \to \mathbb{R}_+ : \exists c \in \mathbb{R}_+ \ \exists n_0 \in \mathbb{N} \ \forall n \geq n_0 : cg(n) \leq h(n)\}$

Since this notation is very cumbersome to work with, we will state a central theorem for computing those runtimes in the case they are described in a recursive manner.

Theorem 1.3 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants and f and g be functions. Moreover, let $T(n)$ be a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $\frac{n}{b}$ is rounded up or down if necessary. Then

(i) $T \in \Theta(n^{\log_b(a)})$, if $f \in \mathcal{O}(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$,

(ii) $T \in \Theta(n^{\log_b(a)} \log(n))$, if $f \in \Theta(n^{\log_b(a)})$ and

(iii) $T \in \Theta(f)$, if $f \in \Omega(n^{\log_b(a)+\epsilon})$ for some $\epsilon > 0$ and $af(nb^{-1}) < f(n)$.

[1] In the following we will abuse this notation and use relations like $=$ and operations like $\cdot$ and $+$ on the sets $\mathcal{O}$, respectively, Ω and Θ as if they were a function contained in that set.

Chapter 2

Convex Hulls

2.1 Definitions

To better understand the problem at hand, we first need to define some basic concepts.

Definition 2.1 (Convex sets)

A set $S \subseteq \mathbb{R}^n$ is called a *convex set* if for any two points $x, y \in S$ the line segment $\overline{xy}$ is fully contained in S.

Thus, convex sets are more or less shapes with no dents, i.e. shapes where no vertex creates an inward pointing corner. Since there are infinitely many convex sets, we need to define a notion of convexity that is more useful in practice.

Definition 2.2 (Convex hull)

The *convex hull* $\text{conv}(S)$ of a set $S \subseteq \mathbb{R}^n$ is the smallest convex set containing S, i.e.

$$\text{conv}(S) := \bigcap_{S \subseteq C \text{ convex}} C.$$

2.2 Applications

Since the convex hull is a fundamental geometric object, it has many applications in computer science, physics, and mathematics. Here are some examples:

1. **Robot motion planning.**
   Problem: Given two points $s, t \in \mathbb{R}^2$ and a polygon P, such that $s, t \notin P$ find a shortest path from s to t that does not intersect P.
   Solution: Calculate a shortest path from s to t along $\partial\text{conv}(P \cup \{s, t\})$, since this is also a shortest path from s to t in $\overline{P^{\complement}}$.

2. **Farthest pair problem.**
   Problem: Given a polygon P, find two points p and q in P that maximize $\|p - q\|_2$.
   Solution: Compute the convex hull of P and return the pair of vertices with largest distance from each other.

To be able to solve these problems, we now want to take a look at different algorithms for computing the convex hull.

2.3 Algorithms

Since we want to compare the different algorithms, we will use the following specifications:

> Given a finite set $S \subseteq \mathbb{R}^2$ consisting of n points, return those contained in $\partial\text{conv}(S)$ in clockwise order.

2.3.1 Naive approach

The first idea that comes to mind is based on the following fact:
A line segment is part of the boundary of the convex hull if and only if there are no points on either the left or the right of it. Moreover, if we traverse the boundary of the convex hull in clockwise direction, then all other points lie to our right at any given moment.
This now motivates our first algorithm.

Algorithm 1 Naive convex hull

Input: A finite set $S \subseteq \mathbb{R}^2$ consisting of n points
Output: Points of S contained in $\partial\text{conv}(S)$ in clockwise order
Runtime: $\mathcal{O}(n^3)$

```
1:  procedure ConvexHull(S)
2:      H ← ∅
3:      for p ∈ S do                                          ▷ O(n^3)
4:          for q ∈ S \ {p} do                                ▷ O(n^2)
5:              valid ← True
6:              for r ∈ S \ {p, q} do                         ▷ O(n)
7:                  valid ← IsRightTurn(p, q, r)
8:              end for
9:              if valid then
10:                 H ← H ∪ {pq}
11:             end if
12:         end for
13:     end for
14:     return InCyclicOrder(H)                               ▷ O(n log(n))
15: end procedure
```

As we can easily see, this algorithm has a cubic runtime, i.e. doubling the input size increases the runtime by a factor of eight. This is not very good, but we can do better.

2.3.2 Graham's scan (Lecture version)

Instead of testing all segments for their containment in the boundary of the convex hull, we could go about this problem in another way. Since the points of S contained in the boundary convex hull are more or less the 'extreme' points in S, we could first split S in an upper and lower half and compute their respective convex hulls by looking for the uppermost, respectively lowermost vertices in both halves and connect them appropriately. Then stitch both halves together to obtain the convex hull of S. This idea is due to Ronald Graham, who published the original algorithm in 1972.

Algorithm 2 Graham's scan

Input: A finite set $S \subseteq \mathbb{R}^2$ consisting of n points
Output: Points of S contained in the convex hull $\text{conv}(S)$ in clockwise order
Runtime: $\mathcal{O}(n \log(n))$

procedure ConvexHull(S)
  $S \leftarrow \text{LexicographicSort}(S)$ ▷ $\mathcal{O}(n \log(n))$
  $H_+ \leftarrow S[0, 1]$
  for $i = 2, \ldots, n-1$ **do** ▷ $\mathcal{O}(n)$ (max. 2 H_+-OPs per point)
    $H_+ \leftarrow H_+ \cup \{S[i]\}$
    while $|H_+| > 2 \wedge \text{IsLeftTurn}(H_+[-3], H_+[-2], H_+[-1])$ **do**
      $H_+ \leftarrow H_+ \setminus H[-2]$
    end while
  end for
  $H_- \leftarrow S[-1, -2]$
  for $i = 3, \ldots, n$ **do** ▷ $\mathcal{O}(n)$ (max. 2 H_--OPs per point)
    $H_- \leftarrow H_- \cup \{S[-i]\}$
    while $|H_-| > 2 \wedge \text{IsLeftTurn}(H_-[-3], H_-[-2], H_-[-1])$ **do**
      $H_- \leftarrow H_- \setminus H[-2]$
    end while
  end for
  $H_- \leftarrow H_- \setminus \{H_-[0], H_-[-1]\}$
  return $H_+ \cup H_-$
end procedure

One should note that there are edge cases which are not yet considered in this algorithm. For example, if three points lie on a line, the algorithm will not remove the middle of the three, which might be undesirable. A remedy for this is to swap the isLeftTurn function with ¬isRightTurn.

2.3.3 Jarvis' march

Another approach is due to R. A. Jarvis, who published it in 1973. This algorithm is widely known as the gift wrapping method. It starts with the lowest leftmost point p_0 and walks around the convex hull in clockwise direction adding the extreme points to the calculated set until it coincides with the convex hull.

Algorithm 3 Jarvis' march

Input: A finite set $S \subseteq \mathbb{R}^2$ consisting of n points
Output: Points of S contained in the convex hull $\text{conv}(S)$ in clockwise order
Runtime: $\mathcal{O}(nh)$

procedure ConvexHull(S)
  $p_1 \leftarrow \text{LowestLeftmost}(S)$ ▷ $\mathcal{O}(n)$
  $H \leftarrow \{(p_1^{(x)}, p_1^{(y)} - 1), p_1\}$
  $q \leftarrow p_0$
  while $q \neq p_1$ **do** ▷ $\mathcal{O}(nh)$
    $q = \text{argmin}_{\substack{p \in S \\ p \text{ right of } H}} \sphericalangle(\overline{H[-2]H[-1]}, \overline{H[-1]p})$ ▷ $\mathcal{O}(n)$
    $H \leftarrow H \cup \{q\}$
  end while
  return $H[1, \ldots, \text{end}]$

10: **end procedure**

A simple analysis shows that the convex hull is computed in $\mathcal{O}(nh)$ time, where h is the number of points in the convex hull. Thus, the runtime of Jarvis' march is dependent on its output. Such algorithms are called *output-size sensitive*.

2.3.4 Further algorithms

Since the publication of Graham's scan and Jarvis' march, many other algorithms have been proposed. The following list is not exhaustive, but gives an overview of the most important ones:

1. **Divide-and-conquer (Preparata & Hong, 1977).**
   Idea: Split S into two subsets S_1 and S_2 and recursively compute the convex hulls of S_1 and S_2. Then, compute the convex hull of the union of S_1 and S_2 by merging the two convex hulls.
   Runtime: $\mathcal{O}(n \log(n))$.

2. **QuickHull (Eddy 1977, Bykett 1978).**
   Idea: Discard non-hull points as quickly as possible.
   Runtime: $\mathcal{O}(n \log(n))$ in favorable case, $\mathcal{O}(n^2)$ in worst case.

3. **Kirkpatrick-Seidel algoithm (1986).**
   Runtime: Optimal output-sensitive runtime of $\mathcal{O}(n \log(n))$.
   Drawback: Very complex.

4. **Chan's algorithm (1989).**
   Idea: Combination of Graham's scan and Jarvis' march.

2.4 Variations on the convex hull problem

Besides the standard convex hull problem, there are several variations on it which are worth mentioning but not of further relevance for the lecture.

1. **Online convex hull algorithms.**
   Problem: Input points are obtained sequentially.
   Observation: Convex hull increases by at most one point each time.
   Optimal runtime: $\mathcal{O}(\log(n))$.

2. **Dynamic convex hull algorithms.**
   Problem: Points can be sequentially inserted or deleted.
   Observation: Removing a point can increase the convex hull's size drastically.
   Optimal runtime: $\mathcal{O}(\log^2(n))$.

3. **kD convex hull algorithms.**
   Problem: Input points are in $\mathbb{R}^k$.
   Observation: Convex hull is a polytope.
   Optimal runtime: $\mathcal{O}(n\lfloor\frac{d}{2}\rfloor)$.

Chapter 3

Line Segment Intersection

3.1 Applications

At first glance it seems that the problem of finding all intersections of line segments is not very useful, since most real life objects are not perfectly straight lines. However, one can use the fact that any curve can be approximated arbitrarily well by a set of line segments, making the algorithms applicable to the following problems:

1. **GIS.**
   Problem: Given a map with roads and a map with rivers, compute all points, where bridges must be constructed.
   Solution: Calculate all intersections of roads and rivers.

2. **Graph intersection.**
   Problem: Given two planar graphs embedded in the plane, compute all their intersections to combine them into a single planar graph.
   Solution: Calculate all intersections of the edges of both graphs, duplicate and shorten them and add new vertices accordingly.

3.2 Algorithms

Again, we want to specify what problem the algorithms should be able to solve:

> Given a finite set S consisting of n line segments embedded in the plane, return all of their intersections.

For simplicity we assume, that no three segments intersect in the same point or that segments are vertical, horizontal or overlapping.

3.2.1 Naive approach

Like before, one first might have the idea to check all segments against each other and return all found intersections. However, this approach is not very efficient, since it has a runtime of $\mathcal{O}(n^2)$, which is undesirable under most circumstances. Nevertheless, it is a good starting point for the development of more efficient algorithms.

Algorithm 4 Naive line segment intersection

Input: a finite set S consisting of n line segments

Output: Intersection points of all segments in S
Runtime: $\mathcal{O}(n^2)$

```
1: procedure FindIntersections(S)
2:     I ← ∅
3:     for s_1 ∈ S do                                   ▷ O(n^2)
4:         for s_2 ∈ S \ {s_1} do                       ▷ O(n)
5:             I ← I ∪ Intersect(s_1, s_2)              ▷ O(1)
6:         end for
7:     end for
8:     return I
9: end procedure
```

3.2.2 Sweep line approach

The most popular algorithm for solving this problem is due to Jon Bentley and Thomas Ottmann, who published it in 1979. It uses the concept of a so-called *sweep line* to only check segments that are close to each other for intersections. To do so, the segments are sorted by their y-coordinates and then a horizontal line is swept from top to bottom. Whenever the line intersects a segment, we add it to a set of active segments. Whenever the line leaves a segment, we remove it from the set of active segments. Whenever two segments in the active queue change their position, they have intersected. Thus, we only need to test neighbors in the active queue for intersections.

Algorithm 5 Sweep line algorithm due to Jon Bentley and Thomas Ottmann

Input: A finite set of n line segements S
Output: All intersections of the segments in S
Runtime: $\mathcal{O}((n + |I|)\log(n))$

```
1: procedure FindIntersections(S)
2:     Q_E ← FillEventQueue(S)                          ▷ O(n)
3:     Q_A ← ∅
4:     I ← ∅
5:     while |Q_E| > 0 do                               ▷ O((n + |I|) log(n))
6:         p ← Q_E.pop()
7:         HandleEventPoint(p, I)                       ▷ O(log(n))
8:     end while
9:     return I
10: end procedure
11: procedure HandleEventPoint(p, I)
12:     s_p ← SegmentOf(p)
13:     if p is upper point then
14:         Q_A.push(s_p)                                ▷ O(log(n))
15:     else if p is lower point then
16:         Q_A ← Q_A \ {s_p}                            ▷ O(log(n))
17:     else if p is intersection point then
18:         (s_i, s_j) ← InvolvedSegments(p)
19:         Q_A ← Q_A.swap(s_i, s_j)                     ▷ O(log(n))
20:     end if
21:     for s_i, s_{i+1} ∈ Q_A do
22:         p_{s_i s_{i+1}} ← Intersect(s_i, s_{i+1})
23:         if p_{s_i s_{i+1}} and p_{s_i s_{i+1}} ∉ I then
24:             I ← I ∪ {p_{s_i s_{i+1}}}
```

```
25:         end if
26:         if p_{s_i s_{i+1}} and p_{s_i s_{i+1}} ∉ Q_E then
27:             Q_E.push(p_{s_i s_{i+1}})                    ▷ O(log(n))
28:         end if
29:     end for
30: end procedure
```

Theorem 3.1

The sweep line algorithm due to Jon Bentley and Thomas Ottmann has a runtime of $\mathcal{O}((n + |I|)\log(n))$.

Proof. We proof by induction over the priority of the event. To do so, we assume that all events with a higher priority have been handled correctly.
Let p be an intersection point and $U(p), L(p)$ and $C(p)$ be the segments, where p is the upper or lower endpoint or a point contained inside the segment, respectively.
Case 1: p is an endpoint. In this case p is stored in Q_E in the beginning. Thus, all segments in $U(p)$ are accessible. Segments in $L(p)$ and $C(p)$ are stored in Q_A when p is reached by the induction assumption.
Case 2: p is an endpoint. In this case all segments containing p are in $C(p)$. These segments are sorted by angle around p. Thus, there are neighboring s_i and s_j intersecting in p. Thus the intersection event in p will be detected.
These two cases cover all possible events. Thus, the algorithm is correct, i.e. all intersections are correctly identified. □

Remark: One can implement the algorithm with $\mathcal{O}(n)$ space-complexity.

Chapter 4

Map Overlay

4.1 Applications

To demonstrate the usefulness of the algorithms presented in this chapter, we will name two applications that usually rely on their existence.

1. **Boolean operations on Polygons.**
   Problem: Given two polygons P_1 and P_2, compute their union, intersection or difference.
   Solution: Overlay both polygons, intersect their edges and label all emerging faces with P_1 or P_2 or both depending on the polygon it is contained in.

2. **Weather data.**
   Problem: Given maps for *amount of precipitation per year* and *hours of sunshine per year* find the regions with more than H hours of sunshine and M mm/m^2 of precipitation a year.
   Solution: Interpret the regions of interest as polygons and apply the approach of the problem above.

4.2 Data structure

It quickly becomes apparent that simply storing our objects as graphs, i.e. a tuple containing vertices and edges, is not feasible since it does not allow for efficient access to corresponding parts like faces, which are necessary for solving our problems. To this end we need a data structure that can efficiently answer the following queries:

(i) Give all edges connected to a vertex in clockwise order.
(ii) Give all boundary edges of a face in clockwise order.
(iii) Give all edges of boundaries of holes in a face in clockwise order.
(iv) Give a face's neighboring face along a given edge.

A data structure providing these operations is the *doubly connected edge list* (DCEL). It is a data structure that represents a planar subdivision of a polygon or the entire plane. It consists of a set of vertices, edges and faces and needs memory in $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}| + |\mathcal{F}|)$. For each of the three we define the following functionals:

Vertices: ▷ Coordinates(v), the 2D-coordinates of the vertex v.
▷ IncidentEdge(v), a pointer to an outgoing half-edge.

Half-edges: ▷ Origin(e), a pointer to the vertex the half-edge starts at.

- ▷ Twin(e), a pointer to the half-edge that is the twin of e, i.e. the half-edge connecting the same vertices but in opposite direction.
- ▷ Next(e) and Prev(e), both pointers to the next and previous half-edge in the face's boundary, respectively.
- ▷ IncidentFace(e), a pointer to the face left to e.

Faces:
- ▷ OuterComponent(f), a pointer to a half-edge on the boundary of the face.
- ▷ InnerComponents(f), a list of pointers to half-edges on the boundaries of holes in the face.

With these constant time operations we can now efficiently execute to following tasks:
Return...

- ▷ the destination of a half edge.
- ▷ all edges connected to a vertex in clockwise order (i).
- ▷ a face's neighboring face along a given edge (iv).
- ▷ all outgoing/ingoing edges connected to a vertex in clockwise order.
- ▷ all vertices adjacent to a given vertex.
- ▷ all vertices incident to a face.
- ▷ all edges that bound a face (ii).
- ▷ all edges of boundaries of holes in a face in clockwise order (iii).
- ▷ all faces that have a given vertex on their boundary.

Remark: People with a background in 3D-modelling might recognize this structure, as it allows for the execution of so-called half-edge collapse methods used in mesh simplification.

4.3 Map overlay algorithm

To be able to state the state the specifications of the algorithm we need to define the following:

Definition 4.1 (Overlay of planar subdivisions)

Let S_1 and S_2 be two planar subdivisions of the plane. Then $O(S_1, S_2)$ is called the *overlay of S_1 and S_2*. $O(S_1, S_2)$ contains a face f if and only if there is a face f_1 in S_1 and a face f_2 in S_2 such that f is a maximal connected subset of $f_1 \cap f_2 \neq \emptyset$.

Thus, our algorithm should be able to do the following:

> Given two planar subdivisions S_1 and S_2 of the plane, compute the overlay $O(S_1, S_2)$ of S_1 and S_2 with each face f in $O(S_1, S_2)$ being labeled with S_1 and/or S_2 depending on where it originated from.

As most parts of the old subdivisions are still part of the overlay, we might as well copy the half-edges of S_1 and S_2 into the data structure D meant to describe $O(S_1, S_2)$.
As this will almost surely create intersections between the edges of S_1 and S_2, we intersect the half-edges in D with the sweep line algorithm from the previous section to get all intersections. All half-edges not intersected by any other can be kept. Intersected edges need to be adjusted in an upcoming step.

To adjust the edges, we only check neighboring edges intersected by the sweep line that do not originate from the same subdivision. Doing so, we can ensure that every part of D above the sweep line is computed correctly. Then update the data involving edges and faces.
For a better understanding of the update steps mentioned above, we will provide rough outlines for the case of an edge and face adjustment:

Outline (Edge adjustment): Suppose an edge e from S_1 passes through a vertex v of S_2. Then:

1. Replace e by edges e_1 and e_2, connecting v and the endpoints of e. This results in four new half-edges.

2. Next, pair up the corresponding twins inside e_1 and e_2.

3. Then fix the next and prev pointers of the new half-edges. All information needed can be acquired by looking at the old half-edges involved.

4. To correct the data stored in v, we first have to compute the cyclic order of the newly created half-edges.

Remark: Since the last step's runtime depends on the degree of v, it is the only step which is not executable in constant time. Overall, the vertex and edge data for the overlay can be computed in $\mathcal{O}((n+k)\log(n))$ time, where n is the compound complexity of S_1 and S_2 and k is the complexity of their overlay.

Outline (Face handling): Since vertices and edges are now computed correctly, it remains to adjust the face data. This is done as follows:

1. For each vertex v in D follow the outgoing half-edges and their successors until they arrive back at v and record the found cycles.

2. For each cycle select the leftmost (and lowest, in case of ties) vertex contained in the cycle and compute the angle α spanned by the cycle half-edges in v.

3. If α is greater than π, the cycle is the boundary of a hole within a face. Otherwise, it is the outer boundary of a face inside the face.

4. Since each outer boundary defines a face, identify all cycles describing outer boundaries[1] with a node in a graph G_D.

5. From each leftmost vertex v of an outer cycle draw a ray to the left and record the first cycle it intersects[2]. Then connect the nodes of the corresponding cycles with an edge in G_D.

6. The connected components of G_D correspond to faces of $O(S_1, S_2)$ and each node within a connected component corresponds to a unique cycle incident to this face. This information can now be used to adjust the inner, respectively, outer component data for each face and the incident face data for each half-edge in D.

With this information, we can now state the algorithm for computing the overlay of two planar subdivisions as layed out above.

[1] To catch the outermost face, one defines a bounding cycle containing all of D.
[2] This has to be an inner cycle, since the ray only moves to the left and thus stays within the face.

Algorithm 6 Map overlay algorithm

Input: Two planar subdivisions S_1 and S_2 as DCELs.
Output: The overlay $O(S_1, S_2)$ of S_1 and S_2 as a DCEL.
Runtime: $\mathcal{O}((n+k)\log(n))$

```
1: procedure MapOverlay(S_1, S_2)
2:     D ← copy of S_1 and S_2                                   ▷ O(n)
3:     I ← Intersect(S_1, S_2)                                   ▷ O((n+k) log(n))
4:     D ← UpdateDWith(I)                                        ▷
5:     C ← ComputeCycles(D)                                      ▷ O(k)
6:     G_D ← ComputeFaceGraph(C)                                 ▷
7:     for c ∈ Components(G_D) do                                ▷
8:         c_outer ← outer boundary cycle of c
9:         c_inners ← inner boundary cycles of c
10:        f ← NewFace()
11:        f.OuterComponent ← c_outer[0]
12:        f.InnerComponents ← [c_inner[0] for c_inner ∈ c_inners]
13:        for n ∈ c do
14:            for e ∈ Cycle(n) do
15:                e.IncidentFace ← f
16:            end for
17:        end for
18:    end for
19:    L ← ComputeFaceLabels()                                   ▷ O((n+k) log(n))
20:    for f ∈ D do                                              ▷ O(k)
21:        f.Labels ← L[f]
22:    end for
23: end procedure
```

Remark: Again, the runtime is output-sensitive and in $\mathcal{O}((n+k)\log(n))$, where n is the compound complexity of S_1 and S_2 and k is the complexity of their overlay. This is not surprising, since the runtime of the algorithm is dominated by the computation of the intersections of the edges in both subdivisions and the labeling of all faces.

Chapter 5

Polygon Triangulation

5.1 The art gallery problem

In this chapter we will focus on one single problem illustrating the wide range of applications of triangulations (cf. definition 11.1 in chapter 11). More precisely, we will consider the *art gallery problem* which is a classical problem in computational geometry. It is stated like this:

Definition 5.1 (The Art Gallery Problem (AGP))

Given a polygon P (representing a set of connected rooms) find a set of points S (representing cameras) such that every point $p \in P$ is connectable to at least one point $s \in S$ such that $\overline{ps} \in P$ (p is visible from s).

To focus on the most important aspects of this problem, we restrict ourselves to simple polygons, i.e. polygons without holes, since the general problem is NP-hard and thus not efficiently solvable, according to today's knowledge.
Giving this problem some thought, one might notice that if P is a simple polygon, the problem can be solved with a set S satisfying $|S| \leq \lfloor \frac{n}{3} \rfloor$[1]. This upper bound arises in the following way: Triangulate the polygon such that each triangle only has vertices which are already in P and 3-color the obtained graph. S can now be defined as the points of P which have the color of least occurrence[2].

5.2 3-coloring of simple polygons

As motivated in the previous section, a first step of solving the AGP is finding a way to 3-color the triangulation of a simple polygon.

Proposition 5.2

Each triangulation T_P of a simple polygon P possesses a 3-coloring.

[1] This bound is sharp. To see this, consider a room looking like a *Toblerone*™ with $n = 3k$ vertices and k spikes.

[2] Since each color occurs in each triangle and triangles are convex, each $s \in S$ can 'see' each point in the triangles containing s. Thus, each point in P is visible from at least one point in S

Proof. Consider the dual graph $G = (V_G, E_G)$ of the triangulation T_P. Then

$$(t_1, t_2) \in E_G \iff \exists (v_1, v_2) =: e \in T_P : e \in t_1 \wedge e \in t_2.$$

Moreover, since P does not contain any holes, G is acyclic and thus a tree.
Now choose an arbitrary $t \in T_P$ and color each vertex a different color. Starting at the node in G corresponding to t, we start a DFS. Since each edge in G connects two triangles sharing an edge in T_P, the triangle corresponding to the node currently visited in the DFS already has two of its vertices colored, i.e. the color for the third vertex is uniquely determined. Thus, we get a 3-coloring of T_P after the DFS visited the last node. □

Remark: The proof of the proposition immediately yields an $\mathcal{O}(n)$ recursive algorithm for computing a 3-coloring of a given triangulation.

5.3 Polygon triangulation

Now that we have an algorithm for 3-coloring a triangulation of a simple polygon, we can turn our attention to the problem of actually finding a triangulation of a simple polygon in the first place.

5.3.1 Naive approach

As before, we start with a naive approach and then try to improve it. The following insight will provide us with a first algorithm.

Proposition 5.3

Each simple polygon with n vertices can be triangulated with $n-2$ triangles each of which only containing vertices in P.

Proof. We show by induction over $n \geq 3$. Since the case $n = 3$ is trivial, we now consider $n > 3$ and assume that the proposition holds for all $3 \leq m < n$.
Let u, v and w be three consecutive vertices of P, where v is the vertex with smallest x-coordinate (and y-coordinate in case of ties). If $\overline{uw} \in P \setminus \partial P$ we can split P along this segment to obtain the partition $P_1 \cup P_2 = P$. Otherwise there is at least one vertex $v \in P$ between v and $\overline{uw}$. Of all such vertices choose the vertex r maximizing the distance to $\overline{uw}$. Then partition P along $\overline{vr}$ into $P_1 \cup P_2 = P$.
Now apply the induction hypothesis to P_1 and P_2 to obtain a triangulation of P_1 and P_2 with $n_1 - 2$ and $n_2 - 2$ triangles respectively and connect them to obtain a triangulation of P with $(n_1 - 2) + (n_2 - 2) = n + 2 - 4 = n - 2$ triangles. □

Remark: The proof of the proposition immediately yields an $\mathcal{O}(n^2)$ recursive algorithm for computing a triangulation of a given simple polygon. Although convex polygons can be triangulated in linear time, partitioning a simple polygon into convex ones again puts the runtime into $\mathcal{O}(n^2)$.

5.3.2 Monotone partitioning

Our first improvement will involve a smarter way of splitting up a simple polygon into a special kind of polygons for which triangulations can be efficiently computed. To formulate this idea in a precise manner, we need to introduce some definitions.

Definition 5.4 (Monotone polygon)

A polygon P is called *g-monotone*, where g is a line in $\mathbb{R}^2$, if

$$\forall g' \perp g \; \exists p, q \in \mathbb{R}^2 : P \cap g' \in \{\emptyset, \overline{pq}\}.$$

To shorten this notation for our most prevalent use cases we will identify the x- and y-axis with the lines $]-\infty, \infty[\times \{0\}$ and $\{0\} \times]-\infty, \infty[$ respectively.

Definition 5.5 (Point ordering)

Let u and v be points. Then $u < v$ (u is below of v), if $u_y < v_y \vee u_y = v_y \wedge u_x > v_x$.

Definition 5.6 (Vertex types)

A vertex v in a y-monotone polygon P with inner angle α and neighbors u and w is called a...

- ▷ **start vertex**, if $u < v$, $w < v$ and $\alpha < \pi$.
- ▷ **split vertex**, if $u < v$, $w < v$ and $\alpha > \pi$.
- ▷ **end vertex**, if $u > v$, $w > v$ and $\alpha < \pi$.
- ▷ **merge vertex**, if $u > v$, $w > v$ and $\alpha > \pi$.
- ▷ **regular vertex**, otherwise.

With these definitions in place, we can now state the following important theorem.

Theorem 5.7 (Characterization of y-monotone polygons)

A polygon P is y-monotone, if it has no split and merge vertices.

Proof. Assume that P iis not y-monotone. Then there is a horizontal line g that intersects ∂P in three points u, v and w from left to right. W.l.o.g. we assume that $\overline{uv} \in P$.
Traversing the polygon starting in v in both directions, the next intersections with g are u and v, respectively. Thus, if $(u, p_1, \ldots, p_r, v)$ contains an end, respectively, start vertex, then the polygon chain $(v, q_1, \ldots, q_r, w)$ must have a split, respectively, merge vertex. Analogously, if $(v, q_1, \ldots, q_r, w)$ contains an end, respectively, start vertex, then the polygon chain $(u, p_1, \ldots, p_r, v)$ must have a split, respectively, merge vertex. Hence, P has at least one split or merge vertex. □

This classification allows us to use a horizontal sweep line moving from top to bottom which triggers and handles an event at each vertex. If the sweep line hits a split or merge vertex, try to connect it with another vertex above or below to produce a partition of P consisting only of y-monotone polygons.

Algorithm 7 Monotone partitioning

Input: A simple polygon P stored as DCEL D.

Output: A partition of P into y-monotone polygons.
Runtime: $\mathcal{O}(n \log(n))$

procedure MonotonePartition(P)
  $Q \leftarrow$ FillQueue(D) ▷ $\mathcal{O}(n \log(n))$
  $T \leftarrow \emptyset$ ▷ $\mathcal{O}(1)$
  while $Q \neq \emptyset$ **do** ▷ $\mathcal{O}(n)$
    $v_i \leftarrow Q$.pop()
    HandleEvent(v_i, Type(v_i), T, D) ▷ $\mathcal{O}(\log(n))$
  end while
  return D
end procedure
procedure StartVertexEvent(v_i, T, D)
  e_i.helper $\leftarrow v_i$
  T.insert((e_i, v_i))
end procedure
procedure EndVertexEvent(v_i, T, D)
  if Type(e_{i-1}.helper) = MERGE **then**
    $v_h \leftarrow e_{i-1}$.helper
    D.insert($\overline{v_i v_h}$)
  end if
  T.remove(e_{i-1})
end procedure
procedure SplitVertexEvent(v_i, T, D)
  $e_j \leftarrow T$.findLeft(v_i)
  $v_h \leftarrow e_j$.helper
  D.insert($\overline{v_i v_h}$)
  e_j.helper $\leftarrow v_i$
  T.insert((e_i, v_i))
end procedure
procedure MergeVertexEvent(v_i, T, D)
  if Type(e_{i-1}.helper) = MERGE **then**
    $v_h \leftarrow e_{i-1}$.helper
    D.insert($\overline{v_i v_h}$)
  end if
  T.remove(e_{i-1})
  $e_j \leftarrow T$.findLeft(v_i)
  if Type(e_j.helper) = MERGE **then**
    $v_h \leftarrow e_j$.helper
    D.insert($\overline{v_i v_h}$)
  end if
  e_j.helper $\leftarrow v_i$
end procedure
procedure RegularVertexEvent(v_i, T, D)
  if IsRightOf(P, v_i) **then**
    if Type(e_{i-1}.helper) = MERGE **then**
      $v_h \leftarrow e_{i-1}$.helper
      D.insert($\overline{v_i v_h}$)
    end if
    T.remove(e_{i-1})
    e_i.helper $\leftarrow v_i$

```
49:         T.insert((e_i, v_i))
50:     else
51:         e_j ← T.findLeft(v_i)
52:         if Type(e_j.helper) = MERGE then
53:             v_h ← e_j.helper
54:             D.insert(v_i v_h)
55:         end if
56:         e_j.helper ← v_i
57:     end if
58: end procedure
```

Remark: In this algorithm we assume, that v_1 is the first vertex (largest vertex with respect to '>') and all other vertices are numbered counterclockwise. Moreover, we set $e_i := \overline{v_i v_{i+1}}$. This pre-processing step does not worsen the overall runtime. Moreover, T stores the (e,v)-pairs ordered by the left-to-right position of e.

Proposition 5.8

MonotonePartition yields a correct partition into y-monotone polygons.

Proof. First, note that all split and merge vertices are removed by upward and downward diagonals, respectively. Here, we only show that SplitVertexEvent produces no intersecting diagonals. All other events are handled similarly.
Let $\overline{v_i v_h}$ be the newly inserted diagonal, e_j and e_k be the two edges left and right of v_i and e_j.helper $= v_h$. Then, consider the area $Q \subset P$ between e_j and e_k bounded by two lines with slope zero, one passing through v_i and one passing through v_h. Since v_h is the last event before v_i relative to e_j, Q does not contain any further edges or vertices. Thus, the added diagonal cannot intersect any other edge in D. □

5.3.3 Triangulation of monotone polygons

Now that we are able to generate a partition of P into y-monotone polygons, we can triangulate each polygon separately and then put the different triangulations back together. To obtain a triangulation of a y-monotone polygon, we will pursue a greedy approach:
Take a y-monotone polygon P and split it into a left and right chain relative to the first (top) vertex. Then greedily connect vertices on opposite chains, if possible.
Naively following this strategy will cause problems at vertices v located at corners with an inner angle of $\alpha(v) > \pi$, since the inserted line might not lie inside P. To bypass this problem, we use a stack S to record all possible vertices. If $v_{\text{last}} := S$.pop(), then a potential diagonal $\overline{v_i v_k}$ is legal in a right/left chain, if v_k is on the left/right of $\overline{v_i v_{\text{last}}}$.

Algorithm 8 Triangulation of monotone polygons

Input: y-monotone polygon P as DCEL D
Output: Triangulation of P in D
Runtime: $\mathcal{O}(n)$

```
1: procedure TriangulateMonotone(P)
2:     V ← Ysort(D)
3:     S ← ∅
4:     S.push(V[0], V[1])
5:     for i = 3, ..., |V| − 1 do
```

```
        v_last ← S.pop()
        if Chain(V[i]) ≠ Chain(v_last) then
            while |S| > 0 do
                D.push(\overline{V[i]v_last})
                v_last ← S.pop()
            end while
            S.push(V[i − 1], V[i])
        else
            v_last ← S.pop()
            while |S| > 0 do
                v_last ← S.pop()
                if IsLegal(\overline{V[i]v_last}) then
                    D.push(\overline{V[i]v_last})
                end if
            end while
            S.push(v_last, V[i])
        end if
    end for
    v_last ← S.pop()
    while |S| > 1 do
        v_last ← S.pop()
        D.push(\overline{V[n]v_last})
    end while
    return D
end procedure
```

5.3.4 Triangulation of simple polygons

Combining these algorithms now allows us to triangulate arbitrary simple polygons by first partitioning the simple polygon into y-monotone polygons, triangulating each of them separately and then gluing their respective triangulations back together.

Algorithm 9 Triangulation of simple polygons

Input: Simple polygon P as DCEL D
Output: Triangulation of P in D
Runtime: $\mathcal{O}(n \log(n))$

```
procedure TRIANGULATE(P)
    D_monotone ← MonotonePartition(P)
    for f ∈ InnerFaces(D_monotone) do
        D.add(TriangulateMonotone(f))
    end for
    return D
end procedure
```

Remark: MonotonePartition also works for polygons with holes. In this case, the total runtime for triangulation is $\Omega(n \log(n))$. Additionally, in 1991, Chazelle developed a $\mathcal{O}(n)$ algorithm for simple polygons. In higher dimensions this problem becomes NP-hard.

Chapter 6

Quadtrees

6.1 Applications

Having discussed various sweep line algorithms, we now want to turn our attention to so-called spatial subdivision data structures. These data structures are used to efficiently answer point or range queries in a given space. In this chapter, we will discuss the point quadtree, a data structure that allows for efficient point queries. Spatial subdivision data structures are used in a variety of applications across multiple fields of study. Here are some examples:

1. **Collision detection.**
   Problem: Given a set of circular objects, determine which objects are in collision.
   Solution: Partition the space into smaller spaces and only check objects in the same or neighboring spaces.

2. **Image compression.**
   Problem: Given an image, try to reduce its size below a given bound.
   Solution: Group regions of similar color. Then color each region with the average color of all pixels contained in that region. Finally, run-length encode the resulting image.

3. **Contrast detection.**
   Problem: Given an image, group regions of low contrast.
   Solution: Consider each pixel as a point with a value associated to it. If the maximal and minimal value differ by more than a given bound b, subdivide the image into four sub-images. Repeat this procedure until the maximal and minimal value differ by at most b or a sub-image only consists of a single pixel.

6.2 Data structure

To represent a spatial subdivision allowing for efficient point queries, we will use a:

Definition 6.1 (Point quadtree)

Let $\sigma := [x_1, x_2] \times [y_1, y_2]$ be a square in the plane and S be a finite set of points in σ. If $|S| \leq 1$, the point quadtree Q_S for S is a single leaf storing σ and S. Otherwise, let

$Q_{\text{NE}}, Q_{\text{NW}}, Q_{\text{SW}}$ and Q_{SE} be the four subtrees of Q_S such that

$$\begin{aligned}
Q_{\text{NE}}.S &= \{(x,y) \in S : x \geq x_{\text{mid}} \wedge y \geq y_{\text{mid}}\} \quad \text{and} \quad Q_{\text{NE}}.\sigma &= [x_{\text{mid}}, x_2] \times [y_{\text{mid}}, y_2]\,, \\
Q_{\text{NW}}.S &= \{(x,y) \in S : x < x_{\text{mid}} \wedge y \geq y_{\text{mid}}\} \quad \text{and} \quad Q_{\text{NW}}.\sigma &= [x_1, x_{\text{mid}}] \times [y_{\text{mid}}, y_2]\,, \\
Q_{\text{SW}}.S &= \{(x,y) \in S : x < x_{\text{mid}} \wedge y < y_{\text{mid}}\} \quad \text{and} \quad Q_{\text{SW}}.\sigma &= [x_1, x_{\text{mid}}] \times [y_1, y_{\text{mid}}] \text{ and} \\
Q_{\text{SE}}.S &= \{(x,y) \in S : x \geq x_{\text{mid}} \wedge y < y_{\text{mid}}\} \quad \text{and} \quad Q_{\text{SE}}.\sigma &= [x_{\text{mid}}, x_2] \times [y_1, y_{\text{mid}}]\,.
\end{aligned}$$

with $x_{\text{mid}} = \frac{x_1+x_2}{2}$ and $y_{\text{mid}} = \frac{y_1+y_2}{2}$. If for any of the four subtrees Q_D it holds that $|Q_D.S| > 1$, the subtree is subdivided again according to the rules above.

Remark: Variants of this definition may allow for larger subtrees, e.g. Q_L may be a single leaf storing σ and S if $|S| \leq 4$.

Looking at this definition, it is immediate that the structure of a point quadtree is strongly dependent on the input. If the input is highly concentrated in a certain region, the quadtree will become badly unbalanced. This can lead to a large number of leaf nodes, which in turn can lead to a large number of comparisons in collision detection or image compression. We thus want to find out, how the depth and other structural properties of a point quadtree depend on the input.

Lemma 6.2 (Bound on the depth of a quadtree)

Let S be a finite set of points in $\sigma := [x_1, x_2] \times [y_1, y_2]$ as above. Then the depth of the point quadtree Q_S is at most $\log(ld_{\text{min}}^{-1}) + \frac{3}{2}$, where $l := x_2 - x_1$ is the sidelength of σ and $d_{\text{min}} := \min_{x,y \in S, x \neq y}(\|x - y\|_2)$ the minimal distance between two points in S.

Proof. W.l.o.g. we assume $l = 1$. Then the sidelength of a square σ on level i is 2^{-i}. Thus the maximal distance of points in a square on level i is $2^{-i}\sqrt{2}$ by Pythagoras' theorem. If $d_{\text{min}} \leq 2^{-i}\sqrt{2}$, a non-leaf square σ contains at least two points. Hence, $i \leq \log(ld_{\text{min}}^{-1}) + \frac{1}{2}$. Accounting for leaves one level below we get the claimed bound. □

Theorem 6.3 (Complexity of quadtrees)

A quadtree of depth d, storing a set of n points, has $\mathcal{O}((d+1)n)$ nodes and can be constructed in $\mathcal{O}((d+1)n)$ time.

Proof. The number of leaves is $N_L = 3N_I + 1$, where N_I is the number of internal nodes. Since an internal node contains at least two points and nodes represent disjoint regions, the total number of internal nodes is n. Since the root node is at depth zero, the claimed relationship holds.
The construction time immediately follows by the linear runtime of the subdivision step. □

6.3 Neighbor search

Since points in different leaves can still be very close to each other, it is very important to be able to determine the neighbor of a given node in a certain direction. Thus, our algorithm should be able to do the following:

Given a quadtree Q, a node v and a direction N/E/S/W, find the node v', such that $Q_v.\sigma$ is a neighbor of $Q_{v'}.\sigma$, where Q_ν is the subtree of Q rooted at ν.

Since adjacent cells in a quadtree can be of different sizes, we do not want to go deeper down the tree than the level of the node we are querying the neighbor of. Thus, we search for a node v' such that $Q_v.\sigma$ and $Q_{v'}.\sigma$ are adjacent and on the same level or on levels above if no such node exists. If this also fails, there is no such neighbor.

Algorithm 10 Neighbor search in northern direction

Input: Node v of a quadtree Q.
Output: Node v' of Q such that $Q_{v'}.\sigma$ is the the north neighbor of $Q_v.\sigma$.
Runtime: $\mathcal{O}(d+1)$

```
 1: procedure FindNorthNeighbor(v, Q)
 2:     if v = Q.root then
 3:         return None
 4:     else if v = v.parent.SW then
 5:         return v.parent.NW
 6:     else if v = v.parent.SE then
 7:         return v.parent.NE
 8:     end if
 9:     μ ← FindNorthNeighbor(v.parent, Q)                    ▷ O(d)
10:     if μ = None ∨ μ.isLeaf() then
11:         return μ
12:     else if v = v.parent.NW then
13:         return μ.parent.SW
14:     else
15:         return μ.parent.SE
16:     end if
17: end procedure
```

Remark: The algorithm can be easily extended to the other directions by replacing the SW/SE/NW/NE checks by the corresponding directions.

Chapter 7

Range- and kD-Trees

7.1 Range Searching

Sometimes, one single point query does not provide us with the full information we need for our task. Thus, it is important to be able to efficiently filter all points we are interested in, without having to query every single point in our data set. This is the task of range searching.

7.1.1 Applications

To get a better understanding of the problem we are trying to solve, some common applications are provided below:

1. **Outlier removal.**
   Problem: Given a set of numbers S and their mean value m. Find all numbers $x \in S$ that not farther away from m than a given bound b.
   Solution: Use a range search to find all numbers $x \in S$ that lie in the 1-dimensional range $[m - b, m + b]$.

2. **Location search.**
   Problem: Given a map with cities C_i and a designated city C, find all cities C_j that are at most 100km away[1] from city C.
   Solution: Use a range search to find all cities C_j that lie in the 2-dimensional range $[C_x - 100, C_x + 100] \times [C_y - 100, C_y + 100]$.

3. **Database querying.**
   Problem: Given a database D with columns $C_D = \{c_1, c_2, \ldots, c_r\}$ and $n >> 1$ entries, find all entries such that the values in columns $(c_i)_{i \in I}$ for some $I \subset \{1, \ldots, r\}$ lie in given ranges $([a_i, b_i])_{i \in I}$ with $a_i < b_i$.
   Solution: Use a r-dimensional range search to find all entries $e \in D$ such that $e \in \bigtimes_{i \in I} [a_i, b_i]$.

As we can see, range searching is not neccessarily limited to 1- or 2-dimensional spaces. In fact, it can be generalized to any number of dimensions. Thus, our algorithms should be able to handle any number of dimensions without having to provide a different approach for every single one.

[1] With respect to the Manhattan metric, i.e. the metric (cf. definition 10.1 in chapter 10) d given by $d(x, y) = \sum_{i=1}^{k} |x_i - y_i|$ with $x, y \in \mathbb{R}^k$.

7.1.2 Algorithms

Having said that, it is time to formalize our task before we can start investigating the problem in more detail trying to develop an efficient algorithm.

> Given a set P of n points in a k-dimensional space and a k-dimensional axis-aligned query range R, find all points in $P \cap R$.

Naive approach

Although we have already mentioned that we do not want to follow this approach, we still state it for completeness' sake.
For each point $p \in P$, we check if $p \in R$ and report p depending on if it is inside the range or not. This obviously yields a $O(n)$ time complexity for any range, which is inefficient if only $k << n$ points lie in R.

Quadtrees

Another approach would be using the quadtrees introduced in the last chapter by constructing a quadtree on top of the set of points and only checking points in grid cells that are intersected by the given query range. Although this may work quite well in some cases, we have already seen that if the points are not evenly distributed, the tree can be very unbalanced leading to many empty cells that need to be checked impacting the runtime of our algorithm.

7.2 kD-trees

To overcome the problems of quadtrees, we can use a different approach to partition the space we are interested in. Instead of partitioning it into four regions each time, we now only create two subspaces enabling us to apply algorithms similar to ones in a binary search tree. This, however, raises the question of how to do this, if multiple dimensions are involved since it is not obvious what these two subspaces should be.

7.2.1 Construction of a kD-tree

To answer this question, we need to construct a variant of a quadtree that is more robust to unbalanced data and enables us to define a way of partitioning space in a way that is easy to understand and easy to implement independent of the number of dimensions involved.
As mentioned above, we want to partition the space of interest in a similar way to how a binary search tree partitions an ordered set of objects. But this time, due to the higher dimensionality of our data, we cycle through the different spatial axis with each new level to sort our data. In two dimension this amounts to the following:
For a node v_x representing the point p, the left subtree v_x.left contains all points p' such that $p'_x \leq p_x$ and the right subtree v_x.right contains all points p' such that $p'_x > p_x$. Moreover, all subtrees of v_x are of the form $\cdot_y$. For a node v_y representing the point p, the left subtree v_y.left contains all points p' such that $p'_y \leq p_y$ and the right subtree v_y.right contains all points p' such that $p'_y > p_y$. Moreover, all subtrees of v_y are of the form $\cdot_x$.

Algorithm 11 Construction of a kD-tree

Input: A set of points $P = \{p_1, p_2, \ldots, p_n\}$ in a 2D space.
Output: A kD-tree T storing the points in P.

Runtime: $\mathcal{O}(n \log(n))$

procedure BUILDKDTREE(P)
  $T \leftarrow$ BuildkDTreeRec(P, x) ▷ $\mathcal{O}(n \log(n))$
  return T
end procedure
procedure BUILDKDTREEREC(P, axis)
  if $|P| = 0$ **then**
    return Empty()
  end if
  $P \leftarrow$ SortByAxis(P, axis) ▷ $\mathcal{O}(n \log(n))$
  $v \leftarrow$ Median(P, axis) ▷ $\mathcal{O}(1)$
  v.left $\leftarrow$ BuildkDTreeRec($P[P.\text{axis} \leq v.\text{axis}] \setminus \{v\}$, NextAxis(axis)) ▷ $\mathcal{O}(\frac{n}{2} \log(\frac{n}{2}))$
  v.right $\leftarrow$ BuildkDTreeRec($P[P.\text{axis} > v.\text{axis}]$, NextAxis(axis)) ▷ $\mathcal{O}(\frac{n}{2} \log(\frac{n}{2}))$
  return v
end procedure

Remark: Taking the median as splitting point, guarantees that the subtrees and thus the whole tree are balanced.

7.2.2 Searching in a kD-tree

Now that we know how to construct a kD-tree, we want to turn our attention to the task of exploiting its structure to efficiently search our space for points in the desired range. To this end, we consider a kD-tree T with n nodes storing points in $\mathbb{R}^2$ in its leaves and a query range $R = [a, b] \times [c, d]$.
Similar to the approach using quadtrees, we start our search at the root node v of T. If v lies in R, we add it to the result. Then, we consider the subtrees v.left and v.right and check if they intersect R and if so, recursively search them until we reach a leaf, whose associated point then gets checked against R and reported accordingly.

Algorithm 12 Range search in a kD-tree

Input: A kD-tree T storing points in a 2D space and a query range $R = [a, b] \times [c, d]$.
Output: A set P of points in T such that $p \in R$.
Runtime: $\mathcal{O}(|T \cap R| + \sqrt{n})$

procedure RANGESEARCH(T, R)
  return RangeSearchRec(T.root, R, x)
end procedure
procedure RANGESEARCHREC(v, R, axis)
  $P \leftarrow \emptyset$
  if $v \in R$ **then**
    $P \leftarrow P \cup \{v\}$
  end if
  if v.isLeaf() **then**
    return P
  end if
  if $\min(R_{\text{axis}}) \leq v.\text{axis}$ **then**
    $P \leftarrow P \cup$ RangeSearchRec(v.left, R, NextAxis(axis))
  end if
  if $\max(R_{\text{axis}}) \geq v.\text{axis}$ **then**
    $P \leftarrow P \cup$ RangeSearchRec(v.right, R, NextAxis(axis))
  end if

18: **return** P
19: **end procedure**

Theorem 7.1

Performing a range search in a kD-tree T with n nodes takes $\mathcal{O}(|T \cap R| + \sqrt{n})$ time.

Proof. Reporting a subtree T_v needs $\mathcal{O}(|T_v \cap R|)$ time. Additionally we need to compute the number of nodes visited in total. For this, consider a single side of the query rectangle. Then, the number of intersected rectangles, hence nodes visited in the algorithm, follows the recurrence

$$Q(n) = \begin{cases} \mathcal{O}(1) & \text{if } n = 1 \\ 2 + 2Q\left(\frac{n}{4}\right) & \text{otherwise} \end{cases}.$$

The solution to this recurrence is $Q(n) \in \mathcal{O}(\sqrt{n})$. Therefore, the total runtime of the algorithm is $\mathcal{O}(|T \cap R| + \sqrt{n})$. □

7.2.3 Higher dimensions

In higher dimensions, we construct a k-dimensional kD-tree by alternate between the k dimensions similar to the 2D case. Since this amounts to more or less the same procedure, we omit the details here and only state the complexity results.

Proposition 7.2

Consider a k-dimensional kD-tree T with n nodes and a query range $R = \bigtimes_{i=1}^{k} [a_i, b_i]$. Then, the following complexities can be achieved:

- ▷ Construction in $\mathcal{O}(n \log(n))$.
- ▷ Range search in $\mathcal{O}(|T \cap R| + n^{1-\frac{1}{k}})$.
- ▷ Memory in $\mathcal{O}(n)$.

7.3 Range-trees

Another approach to efficiently perform range searches is to use a so-called range-tree. It behaves similar to a kD-tree, but is constructed in a different way that we now want to explore.

7.3.1 1D-Range search

For simplicity, we first consider the 1D case and start with a balanced binary search tree (BST) T with n nodes storing numbers in its leaves and a query range $R = [a, b]$.
In contrast to the kD-tree's range search algorithm where we started our search at the root, we now want to find the node v_{split}, where the paths to a and b split, first. After this step, we follow the path from v_{split} to a and report each right subtree at nodes where our path goes to the left. Vice versa for the path from v_{split} to b, we report the left subtree at nodes where our path goes to the right.

Algorithm 13 1D-Range search in a BST

Input: A balanced BST T storing numbers in its leaves and a query range $R = [a, b]$.
Output: All numbers $x \in T$ such that $x \in R$.
Runtime: $\mathcal{O}(|T \cap R| + \log(n))$

```
1: procedure RangeSearch1D(T, a, b)
2:     v_split ← FindSplitNode(T, a, b)                          ▷ O(log(n))
3:     I ← ∅
4:     if v_split.isLeaf() then                                  ▷ O(1)
5:         if v_split.x ∈ R then
6:             I ← I ∪ {v_split}
7:         end if
8:     else
9:         v ← v_split.left
10:        while ¬v.isLeaf() do                                  ▷ O(|T ∩ R| log(n))
11:            if a ≤ v.x then
12:                I ← v.right.report()                          ▷ O(|T ∩ R|)
13:                v ← v.left
14:            else
15:                v ← v.right
16:            end if
17:        end while
18:        v ← v_split.right
19:        while ¬v.isLeaf() do                                  ▷ O(|T ∩ R| log(n))
20:            if b > v.x then
21:                I ← v.left.report()                           ▷ O(|T ∩ R|)
22:                v ← v.right
23:            else
24:                v ← v.left
25:            end if
26:        end while
27:    end if
28:    return I
29: end procedure
30: procedure FindSplitNode(T, a, b)
31:    v ← T.root
32:    while ¬v.isLeaf() ∧ (b ≤ v.x ∨ a > v.x) do
33:        if b ≤ v.x then
34:            v ← v.left
35:        else
36:            v ← v.right
37:        end if
38:    end while
39:    return v
40: end procedure
```

Remark: Getting the data in the needed format requires preprocessing in $\mathcal{O}(n \log(n))$.

7.3.2 2D-Range search

To show how this concept generalizes to higher dimensions, we now want to consider a range-tree in two dimensions. Thus, our task, unsurprisingly, is finding all nodes $x \in T$ such that $x \in R$, given a balanced BST T with n nodes storing points sorted by their x-coordinate in its leaves and a query range $R = [a, b] \times [c, d]$.
The central idea is now to add a pointer for each node v in T that points to an associated BST $T_{\text{assoc}}(v)$ storing the points contained in subtrees of v sorted by their y-coordinate. This is then followed up by performing a 1D-range search on T for the x-range $[a, b]$ and subsequently a 1D-range search on $T_{\text{assoc}}(v)$ for the y-range $[c, d]$ for each v reported in the first step and reporting each point found in this second step.
This nested BST data structure is what we call a *range-tree*.

Algorithm 14 2D-Range search using a range-tree

Input: A range tree T storing points in its leaves and a query range $R = [a, b] \times [c, d]$.
Output: All points $p \in T$ such that $p \in R$.
Runtime: $\mathcal{O}(|T \cap R| + \log(n)^2)$

```
1: procedure RangeSearch2D(T, a, b, c, d)
2:     I ← ∅
3:     v_split ← FindSplitNode(T, a, b)                          ▷ O(log(n))
4:     if v_split.isLeaf() then                                  ▷ O(1)
5:         if v_split.p ∈ R then
6:             I ← I ∪ {v_split}
7:         else
8:             v ← v_split.left
9:             while ¬v.isLeaf() do                              ▷ O(R_T + log(n)^2)
10:                if a ≤ v.x then
11:                    I ← I ∪ RangeSearch1D(T_assoc(v.right), c, d)   ▷ O(R_v + log(n))
12:                    v ← v.left
13:                else
14:                    v ← v.right
15:                end if
16:            end while
17:            v ← v_split.right
18:            while ¬v.isLeaf() do                              ▷ O(R_T + log(n)^2)
19:                if b > v.x then
20:                    I ← I ∪ RangeSearch1D(T_assoc(v.left), c, d)    ▷ O(R_v + log(n))
21:                    v ← v.right
22:                else
23:                    v ← v.left
24:                end if
25:            end while
26:        end if
27:    end if
28:    return I
29: end procedure
```

Remark: For brevity we used the notations $R_T := |T \cap R|$ and $R_v := |T_{\text{assoc}}(v) \cap R|$. Moreover, one can see that in contrast to the kD-tree search algorithm, the comparisons for the x- and y-coordinate are subsequently executed and not in alternating manner

Remark: Given a set of n points in the plane, a range-tree can be constructed in $\mathcal{O}(n \log(n))$ time using $\mathcal{O}(n \log(n))$ memory, since each level has $\mathcal{O}(n)$ nodes and the tree itself has $\mathcal{O}(\log(n))$ levels. This can be done by precomputing two lists, one sorted by x- the other by y-coordinate and then building the range tree from the bottom up.

7.3.3 Higher dimensions

As before, the construction and searching algorithms in higher dimensions proceed analogously to the one in two dimensions, which is why we again omit the details and only state the complexity results.

Proposition 7.3

Consider a k-dimensional kD-tree T with n nodes, $k-1$-dimensional associated range-trees and a query range $R = \bigtimes_{i=1}^{k} [a_i, b_i]$. Then, the following complexities can be achieved:

- ▷ Construction in $\mathcal{O}(n \log(n)^{k-1})$
- ▷ Range search in $\mathcal{O}(|T \cap R| + \log(n)^k)$
- ▷ Memory in $\mathcal{O}(n \log(n)^{k-1})$

Chapter 8

BSP-trees

8.1 Applications

For some applications it is required to store a number of non-zero-dimensional objects, i.e. lines, polygons, polyhedra, etc. in a data structure that allows for efficient queries of their relative position to a point in space that may change over time. The following examples show how this problem arises in the day to day tasks of modern computers:

1. **3D Graphics.**
   Problem: Given a scene with a number of objects, determine which objects are visible from a given viewpoint and render them accordingly.
   Naive approach: Use a frame-/z-buffer to apply the painter's algorithm. This works very well, since we do not need to sort the objects from back to front[1], which makes it independent of the scene's complexity and today's GPUs are excellent at doing these tasks. On the other hand, transparent objects still make problems if they are not rendered in the correct order. Thus, sorting is still required.
   Solution: Compute a partition of the scene into a number of convex subscenes. The subscenes are then rendered in a back-to-front order.

2. **Camera culling.**
   Problem: Given a number of objects (polygons) in a scene, determine which objects are outside of the cameras view and do not render them.
   Solution: Check each object's bounding box against the camera's frustum using a space partition and render those visible in appropriate order.

8.2 Data structure

Since our objects are now more complex than points, we need to design a new data structure. This data structure should be able to do the following:
Partition an n-dimensional space in a way that makes it feasible to dynamically order the objects contained in that space back-to-front when viewed from any given direction.
Again, we fall back on using a binary tree that partitions the space into subspaces that, at their lowest level, each contain at most one object or a fragment of one. The inner nodes store the splitting hyperplane, the objects lying within that hyperplane and possess two children, one for each generated subspace. The leaves partitioning the space now contain the fragments of the objects that lie within the region of the partition they represent.

[1] In some cases objects might cyclicly overlap, which requires them to be cut into multiple sub-objects that can be ordered.

Each of the partitioning hyperplanes h is defined by a normal vector h_n and a point h_p on the hyperplane and divide the space into an 'inside' $(+)$ and 'outside' $(-)$ region defined by

$$h^+ := \{q \in \mathbb{R}^n : \langle q - h_p, h_n \rangle > 0\} \quad \text{and} \quad h^- := \{q \in \mathbb{R}^n : \langle q - h_p, h_n \rangle < 0\}.$$

The hyperplanes are (usually) chosen to be the edges (in 2D, faces in 3D) of the objects within the space.

Proposition 8.1

A binary space partition tree (BSP-tree) with m nodes can, given a view direction z, be traversed in $\mathcal{O}(m)$ time to determine the correct order of the objects within the space.

Proof. The tree can be traversed in a depth-first manner. At each node, the view direction z is compared to the normal vector n_h of the hyperplane h. If the view direction is in the same direction as the normal vector, i.e. $\langle n_h, z \rangle > 0$, the 'inside' region is traversed first, then h and finally h^-, otherwise the 'outside' region is traversed first, then h and then h^+. This is done recursively until the leaves are reached. Since the leaves contain at most one object or fragment, the described traversal yields a back-to-front sorting of the objects. □

8.3 Construction

As for the previous space partitioning data structures, the size of the BSP-tree depends heavily on the input provided for its construction. This time however, it is the order the subspaces are generated in and not the overall distribution of the objects in space that impact the size of our BSP-tree.
As the combinatorics in the case of higher dimensional objects is way more complicated than for points, we use the 'laziest' way out, namely using a random order to achieve a good expected runtime. This approach is called *auto-partitioning*.

Algorithm 15 Construction of a BSP-tree using auto-partitioning

Input: A set $S = \{s_1, \ldots, s_n\}$ of non intersecting line segments in $\mathbb{R}^2$ in random order.
Output: The root v of a BSP-tree T
Runtime: $\mathcal{O}^*(n^2 \log(n))$

```
1:  procedure BSP2D(S)
2:      v ← Node()
3:      if |S| ≤ 1 then
4:          v_S ← S
5:      else
6:          h ← SplitAlong(s_1)
7:          v_S ← {s ∈ S : s ⊂ h}
8:          v_left ← BSP2D({s ∈ S : s ∩ h⁻ ≠ ∅})
9:          v_right ← BSP2D({s ∈ S : s ∩ h⁺ ≠ ∅})
10:     end if
11:     return v
12: end procedure
```

Remark: This algorithm can be analogously extended to higher dimensions by using the edge-analogue of the objects instead of their actual edges. Nowadays, the 3D version of this algorithm is used in many video games to render the scenes dynamically from the player's point of view.

Theorem 8.2

The algorithm BSP2D runs in $\mathcal{O}^*(n^2 \log(n))$ expected time.

Proof. Since the algorithm is non-deterministic we want to calculate the expected runtime. Let s_i be a fixed segment in S. Then a segment s_j is cut by s_i if and only if there is no other segment s_k between s_i and s_j and s_i is directed at s_j. We now want to estimate the probability of this 'shielding' happening. To do this, we define

$$d_i(j) := \begin{cases} |\{s \in S : \text{Ext}(s_i) \cap s \neq \emptyset \wedge s \text{ between } s_i \text{ and } s_j\}| & \text{if } s_i \cap s_j \neq \emptyset \\ \infty & \text{otherwise} \end{cases},$$

where $\text{Ext}(s)$ denotes the extension to an infinite line of a segment s[2]. Now let $k := d_i(j)$ and let $s_{j_i}, \ldots, s_{j_k}$ be the intersected segments between s_i and s_j. Moreover, for s_i to split s_j it has to be chosen as a splitting segment before s_j and $s_{j_i}, \ldots, s_{j_k}$. Thus, $i = \min(i, j, j_1, \ldots, j_k)$. Hence

$$\mathbb{P}(\text{Ext}^*(s_i) \cap s_j \neq \emptyset) \leq \frac{1}{2 + d_i(j)} = \frac{1}{2+k},$$

since there might be $s \in S$ such that $\text{Ext}^*(s)$ shields s_j from s_i. Thus,

$$\begin{aligned} \mathbb{E}[|\{s \in S : s_i \text{ causes split of } s\}|] &\leq \sum_{j \neq i} \frac{1}{2 + d_i(j)} \\ &\leq 2 \sum_{k=0}^{n-2} \frac{1}{2+k} \\ &\leq 2 \log(n), \end{aligned}$$

since $H_n \leq \gamma + \log(n) + \mathcal{O}(\frac{1}{n})$, where H_n is the n-th harmonic number and γ is the Euler-Mascheroni constant. By linearity we thus have

$$\mathbb{E}[\text{Total cuts in BSP2D}] \leq 2n \log(n),$$

hence executing BSP2D leads to at most $n + 2n \log(n)$ fragments and finishes the construction in $n\mathcal{O}^*(n \log(n)) = \mathcal{O}^*(n^2 \log(n))$ expected time. □

Remark: In practice the number of fragmentations is far below the expected value, making the algorithm much more efficient and thus applicable. Moreover, in production, BSP-trees are computed off-line, i.e. before shipping the product, such that it does not need to be computed by the client itself but only loaded from a file. Additionally, one can first use so-called free splits, i.e. segments that do not cause fragmentation of other objects, to reduce the total size of the BSP-tree[3].

Remark: Not using auto-partitioning, i.e. providing the splitting lines as input for the construction of the BSP-tree, allows for an $\mathcal{O}(n \log(n))$ deterministic algorithm, which is also the

[2]Similarly $\text{Ext}^*(s)$ is the maximal extension of s, i.e. the line through s stopping at its first intersection with another segment/extension in each direction.

[3]For example a line segment crossing an entire region of a partition is a free split, since it naturally divides the partition into two smaller subpartitions without fragmenting any segments within the partition itself. Those segments can be identified by initially labeling the endpoints of all segments with 0, and, if they are fragmented by other segments in prior construction steps, labelling the arising split points with 1. Then, if a fragment has endpoints that are both labeled with 1 it can be used as a free split in the next step.

theoretical lower bound. In the 3D worst case the space requirements increase to $\mathcal{O}(n\sqrt{n})$ ($\Omega^*(n^2)$ with auto-partitioning) since one can use a scene with two sets of rectangles that are parallel within each set but orthogonal to the other set.

Chapter 9

Point Location

9.1 Applications

Recall the problem stated in chapter 7 where we wanted to find points within a given region. This time, we want to solve the inverse problem: Given a point, find the region containing it. Two applications relying on a efficient solution to this problem are:

1. **Geo-coordinates to country.**
   Problem: Given a set of long-lat-coordinates, find the country in which this coordinates are located in.
   Solution: Use a space partition that efficiently searches the countries near that coordinates until a country is found that contains them.

2. **GUI interaction.**
   Problem: Given a graphical user interface (GUI) with buttons and a mouse-click event, decide wether a button was clicked and execute its associated action.
   Solution: Partition the GUI in a way that the buttons near the given position can be efficiently searched for a click.

9.2 Definitions

Prior to further investigation of this problem and the development of a solution, we need to fix some notation first.

Definition 9.1 (Non-crossing line segments)

Two segments s_1 and s_2 are *non-crossing* if they do not intersect and or the intersection is a common endpoint.

Definition 9.2 (General position)

A set S of n non-crossing segments whose endpoints are in x-general position[1]is called a set of *line segments in general position*.

[1]Meaning that no points share the same x-coordinate. This can be achieved by shearing the coordinate system. As this would require a lot of unnecessary computations, this is usually done symbolically by sorting

Definition 9.3 (Extensions and bounding box)

Let R be a rectangle containing all of S. Then R is called a *bounding box* of S. Moreover, for each endpoint p of a segment $s \in S$ we define $E^+(p)$ and $E^-(p)$ as the upper, respectively, lower *extension* of p, i.e. the vertical line segments extending upwards, respectively, downwards from p until they either hit another segment $s' \in S$ or the bounding box R.

These definition now allow us to introduce the object central to this chapter, the trapezoidal map.

Definition 9.4 (Trapezoidal map)

Let S be a set of line segments in general position. Then, the *trapezoidal map* $T(S)$ is the subdivision of the plane induced by S, a bounding box R and the set of upper and lower extensions of all endpoints of segments in S, i.e. $\bigcup_{v \in S} \{E^+(v), E^-(v)\}$.

Definition 9.5 (Faces, sides and vertices)

A *face* in a trapezoidal map $T(S)$ is a region bounded by a number of edges in $T(S)$. Segments of maximal length bounding a face form the *sides* of said face. Finally the meeting point of two sides is called a *vertex*.

As these definitions do not specify any spatial relationship between the trapezoids, we need to define the final two concepts before continuing with the actual development of a data structure and algorithm.

Definition 9.6 (Adjacency)

Two faces in $T(S)$ are called *adjacent* if they share a vertical edge.

Definition 9.7 (top, bottom, leftp and rightp)

Consider a face Δ in a trapezoidal map $T(S)$. Then, Δ.top and Δ.bottom refer to the upper, respectively, lower non-vertical sides of Δ. The left and right side of Δ are either a vertical side or a vertex, in case Δ is a degenerate trapezoid, i.e. a triangle. In both cases we can uniquely (since the points are in x-general position) define Δ.leftp and Δ.rightp, referring to the vertices in $T(S)$ lying within the left, respectively, right side of Δ.

9.3 Data structure

To see why the trapezoidal map is a good choice for the problem at hand, we first need to understand what problems arise if we approach this problem in a naive way. But first, let us

the points lexicographically.

specify the problem more formally:

> Given a tesselation of the plane S in general position and a point q, find the polygon $P \in S$ such that $q \in P$.

9.3.1 Naive approach

A first approach would be to insert a vertical line for each vertex $v \in S$ creating a set of 'slabs' consisting of trapezoidal pieces. Then, one would look for the slab containing q and find the polygon P that contains q by checking each trapezoidal pice within that slab.
Although this may seem like a promising attempt on solving this problem, one can come up rather quickly with an example where this approach breaks down. To this end, consider a scenario where S contains $\frac{n}{4}$ (wider than tall) rectangles stacked on top of each other, where one of the rectangles is horizontally split into two halves by a jagged polygon chain consisting of $\frac{n}{4} + 1$ vertices. Then there are a total of $\frac{n}{4}$ slabs each consisting of $\frac{n}{4} + 2$ trapezoidal pieces. Thus, the naive approach has a worst-case running time and memory requirement of $\mathcal{O}(n^2)$ which is undesirable.

9.3.2 Trapezoidal map

To reduce the memory consumption arising from naively partitioning our space into vertical slabs, we can enclose the tesselation by a bounding box R and use a partition that uses vertical lines for each vertex $v \in S$ that only extend to the bounding box or another edge within S. To avoid edge cases we may further assume that the points are in x-general position. This now motivates the usage of the trapezoidal map $T(S)$ that we introduced in the last section. Moreover, we want to provide some facts that will be useful for the analysis of the algorithms developed in the upcoming section.

Lemma 9.8

A trapezoidal map $T(S)$ of a planar tesselation S with n edges in general position has at most $6n + 4$ vertices and $3n + 1$ trapezoids.

Proof. Via induction over n and the maximal number of trapezoids that can share the same left point. □

Lemma 9.9

A trapezoid Δ in a trapezoidal map $T(S)$ in general position has at most four adjacent trapezoids.

Proof. Since the trapezoidal map is in general position, no two vertices can have the same x-coordinate. Thus, the left and right vertical sides of Δ contain at most one vertex splitting the side into at most two edges. If one of the sides is degenerate, i.e. only a single vertex v, there can be at most two further trapezoids containing that vertex. Thus the number of adjacent trapezoids is at most four. □

Remark: A trapezoidal map is implemented in a way such that each trapezoid Δ has a pointer to the points Δ.leftp and Δ.rightp, the edges Δ.top and Δ.bottom and the (at most) four adjacent trapezoids. This allows for the retrieval Δ's geometry in constant time.

9.4 Algorithms

Before we can solve our original problem of identifying the polygon $P \in S$ containing a point q, we need to construct a trapezoidal map $T(S)$ for the tesselation S and find a way of efficiently traversing it to find the trapezoid containing q. Only then will we be able to find the polygon $P \in S$ containing q.

9.4.1 Construction of a trapezoidal map

Before we start constructing the trapezoidal map, we should think about its use in the original searching-problem. To do so, we assume that we already are provided with a trapezoidal map for our tesselation S. If we want to find the trapezoid containing our point of interest q, we start at the root of the trapezoidal map $T(S)$ and alternatingly check if q lies to the left or the right of a vertical line or above, respectively, below the currently selected line segment in S.
This process now raises the question of how to construct a data structure that allows to efficiently execute this search-algorithm efficiently in the first place.
As this problem-class seems familiar by now, it might not be a surprise to see spatial subdivisions showing up again. Hence, we will proceed by using a search data structure similar to a kD-tree by using x-nodes pointing to vertices of S and y-nodes pointing to edges, or rather line segments, in S. Contrary to kD-trees however, these nodes do not need to be traversed in a fixed order. Thus, our search data structure won't be a tree but a directed acyclic graph (DAG). Similar to the BSP-trees, the order in which these nodes are created is heavily dependent on the order the segments are inserted. Hence, we again rely on using a random insertion order. If we can guarantee that after the i-th insertion the search data structure D for the tesselation $S_i := \{s_1, \dots, s_i\}$ is correct then the data structure will be correct for S by an inductive argument.

Algorithm 16 Construction of a trapezoidal map

Input: A set of n non-crossing edges S in general position.
Output: A trapezoidal map $T(S)$ of S and the corresponding search data structure D.
Runtime: $\mathcal{O}^*(n \log(n))$

```
1: procedure TrapezoidalMap(S)
2:     R ← ComputeBoundingBox(S)
3:     T ← Node(R)
4:     D ← Node(R)
5:     S* ← Shuffle(S)
6:     for i = 1, ..., n do
7:         Δ ← IntersectedTrapezoids(s_i*, T, D)
8:         Update(s_i*, Δ, T, D)
9:     end for
10:    return T, D
11: end procedure
12: procedure IntersectedTrapezoids(s_i*, T, D)
13:    p ← s_i*.leftp
```

[1] Although not explicitly stated, we will also use the leftp, respectively, rightp pointer for line segments.

```
    q ← s_i^*.rightp
    Δ_0 ← D.query(p)
    j ← 0
    while Δ_j.rightp = q do
        if Δ_j.rightp > q then
            Δ_{j+1} ← Δ_j.lowerright
        else
            Δ_{j+1} ← Δ_j.upperright
        end if
        j ← j + 1
    end while
    return Δ_0, ..., Δ_j
end procedure
procedure UPDATE(s_i^*, Δ, T, D)
    p ← s_i^*.leftp
    q ← s_i^*.rightp
    Δ_0, ..., Δ_k ← Δ
    if p ∈ Δ_0 then
        D.remove(Δ_0)
        D.insertX(p, Δ_0, NewTrapezoid(p, Δ_0), left)
    end if
    if q ∈ Δ_k then
        D.remove(Δ_k)
        D.insertX(q, Δ_k, NewTrapezoid(q, Δ_k), right)
    end if
    for j = 0, ..., k do
        D.remove(Δ_j)
        D.insertY(s_i^*, Δ_j, Split(s_i^*, Δ_j))
    end for
    D.joinTrapezoids()
    T.updatePointers(D)
end procedure
```

Remark: In the procedure IntersectedTrapezoids it is possible to query the search data structure D for the trapezoid Δ_0 that contains q in the i-th step, since D is already correctly constructed for the previous segments s_j^* with $j < i$[2]. If, however, p exists as endpoint of another segment already inserted into T, i.e. on an x- or y-node in D, one has to consider the following cases:

1. p lies on an x-node: In this case the search for Δ_0 is continued on the right subtree of the x-node.

2. p lies on a y-node: In this case p has to be the endpoint of a segment $s_j^* \in S^*$ with $j < i$. Here, one must consider the slope of the segments involved. If the slope of s_i^* is larger than the slope of s_j^* then the search for Δ_0 is continued in the upper subtree of the y-node. Otherwise, the search is continued in the lower subtree.

Moreover, in the Update procedure one should integrate the pointer updates in T into the search data structure update to improve the runtime of the algorithm to one in $\mathcal{O}(k)$.

[2]In the beginning D only has one leaf, namely the bounding box R which must contain q.

Theorem 9.10

The algorithm TrapezoidalMap generates a trapezoidal map $T(S)$ with expected memory requirements in $\mathcal{O}^*(n)$ and allows for point queries in $\mathcal{O}^*(\log(n))$ expected time.

Proof. First we show the statement for point queries. Let p be a point to be queried and let X_i denote the number of nodes by which the search path to find p grows when inserting s_i^* into T, and hence D. The expected length of the complete search part thus is given by

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

which we now want to calculate. For each additional segment added to T, respectively, D, the search path length grows by at most three, since the algorithm adds no more than three levels to the search data structure at any given point. Thus, we have $X_i \leq 3$ and $\mathbb{E}[X_i] \leq 3P_i$, where P_i is the probability for $X_i > 0$. We now want to bound P_i. For this, note that the search path to p only grows in the i-th step, if the trapezoid containing p is changed by the insertion of s_i^*. This happens if and only if s_i^* determines a side of the trapezoid containing p after the splitting has happened. Because the set $\{s_0, \ldots, s_i\}$ has i edges and all are equally likely to be s_i^*, each of these four cases has a probability of at most $\frac{1}{i}$, hence $P_i \leq \frac{4}{i}$. This yields

$$\mathbb{E}\left[\sum_{i=1}^{n} X_i\right] \leq \sum_{i=1}^{n} 3P_i \leq 12 \sum_{i=1}^{n} \frac{1}{i} = 12H_n \leq 12\log(n).$$

This shows the first part of the theorem.
Due to a previous lemma we know that the number of trapezoids in $T(S_i)$ with $S_i := \{s_1^*, \ldots, s_i^*\}$ is bounded by $3i+1$. The size of D now depends on the number of trapezoids and inner nodes. To estimate the number of inner nodes after the insertion of s_i^* we denote the number of new trapezoids after the insertion of this segment by k_i and immediately get that the number of new inner nods in step i is $k_i - 1$. Thus, the expected memory usage of D is given by

$$\mathcal{O}(n) + \mathbb{E}\left[\sum_{i=1}^{n}(k_i - 1)\right] = \mathcal{O}(n) + \sum_{i=1}^{n} \mathbb{E}[k_i].$$

To determine $\mathbb{E}[k_i]$ for $\Delta \in T(S_i)$ and $s \in S_i$ we define

$$\delta(\Delta, s) := \begin{cases} 1 & \text{if } \Delta \text{ disappears from } T(S_i) \text{ when } s \text{ is removed} \\ 0 & \text{otherwise} \end{cases}.$$

Because there are at most four segments causing Δ to disappear, we obtain

$$\sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq 4|T(S_i)| = \mathcal{O}(i).$$

Averaging over the expected number of inserted trapezoids k_i now yields

$$\mathbb{E}[k_i] = \frac{1}{i} \sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq \frac{\mathcal{O}(i)}{i} = \mathcal{O}(1).$$

Thus, the expected number of new inner nodes is constant for each step resulting in an overall expected memory requirement of $\mathcal{O}^*(n)$. □

Although we have not stated it explicitly, the previous theorem assumed that the algorithm computes the trapezoidal map $T(S)$ correctly. This however was not shown yet. Thus, we have to argue that the algorithm indeed efficiently generates a correct trapezoidal map before concluding this section.

Proposition 9.11

The algorithm TrapezoidalMap correctly generates a trapezoidal map $T(S)$ and its associated search data structure D with an expected runtime in $\mathcal{O}^*(n \log(n))$.

Proof. Since correctness already follows from the loop-invariant, we only need to show the promised runtime complexity.
Accounting for the initialization, the permutation of the segments in the beginning and the repeated search for Δ_0 and insertions of k_i trapezoids, respectively, $k_i - 1$ inner nodes we get a total expected runtime of

$$\mathcal{O}(1) + \mathcal{O}(n) + \sum_{i=1}^{n} (\mathcal{O}^*(\log(i)) + \mathcal{O}(\mathbb{E}[k_i])) = \mathcal{O}^*(n \log(n)).$$

This now concludes the proof. □

Remark: Since the analysis of the algorithm's runtime is based on the random ordering of the segments, the expected runtime does not depend on the randomness of the input itself. Thus, there are no tessellations S for which the runtime is significantly better than expected.

9.4.2 Face search

Now that we have the necessary search data structure and the algorithm to generate a trapezoidal map encompassing it, we can finally solve the problem that initially motivated the construction of this data structure.
As this procedure is not very complicated but rather included for completeness' sake, we will not give a detailed proof. Instead, we will only give a short outline of the algorithm:

Outline (FaceSearch):

1. Generate the trapezoidal map $T(S)$ and its associated search data structure D.
2. Find the trapezoid Δ containing p by performing a point query in D.
3. Use Δ.top and Δ.bottom to determine the face containing Δ and thus p.

Chapter 10

Voronoi Diagrams

10.1 Definitions

Since this chapter introduces some new mathematical concepts, we will first some definitions to properly understand them.

Definition 10.1 (Metric)

A *metric* on a set X is a function $d : X \times X \to \mathbb{R}$ satisfying the following properties:

1. $d(x, y) \geq 0$ for all $x, y \in X$,
2. $d(x, y) = 0$ if and only if $x = y$,
3. $d(x, y) = d(y, x)$ for all $x, y \in X$, and
4. $d(x, y) \leq d(x, z) + d(z, y)$ for all $x, y, z \in X$.

In this lecture we will only consider the metric d induced by the Euclidean norm on $\mathbb{R}^2$, i.e.

$$d(x, y) = \|x - y\|_2 = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}.$$

As we will see later, this will give us the opportunity to derive explicit expressions for some geometric objects required for developing our algorithms. For general, more complex, metrics, one would have to resort to more advanced mathematical tools or even numerical techniques, since the implicit equations involved in our derivation are not analytically solvable with means available to us, if at all possible.

Definition 10.2 (Voronoi diagram)

Let $P = \{p_1, \ldots, p_n\}$ be a finite set of n points in $\mathbb{R}^2$ and d a metric. A tesselation of the plane in n Voronoi cells $V(p_i)$ with

$$V(p_i) := \{x \in \mathbb{R}^2 \mid d(x, p_i) < d(x, p_j) \text{ for all } j \neq i\}$$

is called the *Voronoi diagram* $\mathrm{Vor}_d(P)$ of P with respect to d. The vertices and edges in this diagram are called Voronoi vertices, respectively, Voronoi edges. The faces are called Voronoi cells.

Definition 10.3 (Bisector)

The *bisector* m_{ij} of two points p_i and p_j is the set

$$m_{ij} := \{x \in \mathbb{R}^2 \mid d(x, p_i) = d(x, p_j)\}.$$

Remark: In the case of the Euclidean norm the m_{ij} are straight lines.

10.2 Applications

As with every new problem, we want to motivate its relevance by listing some applications.

1. **Closest hospital.**
   Problem: Given a map with locations of n hospitals $H := \{h_1, \ldots, h_n\}$, find the closest hospital for any point on the map.
   Solution: Compute the Voronoi diagram $\mathrm{Vor}_d(H)$ and for each point p on the map find the Voronoi cell $V(h_i)$ containing p. The d-closest hospital to p then is the corresponding h_i.

2. **Aerospace emergency system.**
   Problem: Given a map with locations of n airports $A := \{a_1, \ldots, a_n\}$ and a set of planes P, find, in case of a widespread emergency, the d-closest airport for each plane in P.
   Solution: Compute the Voronoi diagram $\mathrm{Vor}_d(A)$ and for each plane p find the Voronoi cell $V(a_i)$ containing p. The closest airport to p then is the corresponding a_i.

10.3 Analysis

Now that we have introduced the basic concepts, we can start to analyze the Voronoi diagram in some more detail. We will start with some basic properties and then move on to more advanced results in the upcoming sections.

Remark: Because the bisectors m_{ij} for the euclidean norm are straight lines, the Voronoi cells are intersections of open half-spaces H_{ij}, i.e.

$$V(p_i) = \bigcap_{j \neq i} H_{ij} \quad \text{with} \quad H_{ij} := \{x \in \mathbb{R}^2 : d(x, p_i) < d(x, p_j)\}.$$

Proposition 10.4

Let P be a set of $n \geq 3$ distinct points. If all points lie on a line g then all bisectors are parallel and infinite lines orthogonal to g. Otherwise, the edges in $\mathrm{Vor}(P)$ form a connected graph with each edge being a segment or half-line.

Proof. First, we note that each $V(p_i)$ must be convex since it is the intersection of a number of convex half-spaces. Since the case where all points lie on a line is trivial, we assume that there exist p_i, p_j and p_k that do not lie on a common line. Because all edges in $\mathrm{Vor}(P)$ are pieces of bisectors, they can only be line segments, half-lines or full lines[1]. On the other hand,

[1] To make this a proper graph, introduce a dummy vertex v_∞ connecting to all infinite lines.

the bisectors m_{ij} and m_{ik} must intersect, implying that they cannot completely correspond to edges in $\mathrm{Vor}(P)$. Therefore, full lines are not possible.
For the second statement we may assume that $\mathrm{Vor}(P)$ is not connected. Then there must be a cell $V(p_i)$ separating two components. Consequently, this cell must be bounded by two parallel lines due to convexity of $V(p_i)$. This contradicts the statement shown prior implying that $\mathrm{Vor}(P)$ must be connected. □

Lemma 10.5

A planar Voronoi diagram of $n \geq 3$ points has at most $n_v = 2n-5$ vertices and $n_e = 3n-6$ edges.

Proof. In the case where all points in P lie on a common line g, we already know that $\mathrm{Vor}(P)$ consists out of $n-1$ parallel lines. Connecting those to v_∞ results in a graph with $n-1$ looping edges from v_∞ to v_∞. In the other case we obtain a connected, planar embedded graph by the previous proposition. Using Euler's formula $|V| - |E| + |F| = 2$, we see that

$$2 = (n_v + 1) - n_e + n$$

because $n_f = n$ and introducing v_∞ adds one vertex. Because every edge contributes to two vertices and each vertex has at least three edges, we get $3(n_v + 1) \leq 2n_e$ and in particular

$$\begin{aligned} 6 = 3(n_v+1) - 3n_e + 3n \leq -n_e + 3n &\iff n_e \leq 3n-6 \text{ and} \\ 4 = 2(n_v+1) - 2n_e + 2n \leq -(n_v+1) + 2n &\iff n_v \leq 2n-5. \end{aligned}$$

This proves the proposition. □

10.4 Algorithms

As before, we will start by specifying the problem our algorithm should be able to solve:

Given a set of n points $P = \{p_1, \ldots, p_n\}$ in $\mathbb{R}^2$, compute the Voronoi diagram $\mathrm{Vor}_d(P)$ of P with respect to the Euclidean metric d.

10.4.1 Naive Voronoi

Using the remark above, one might come up with the following algorithm:

Algorithm 17 Naive Voronoi

Input: Set of n distinct points $P = \{p_1, \ldots, p_n\}$.
Output: Voronoi diagram $\mathrm{Vor}_d(P)$.
Runtime: $\mathcal{O}(n^2 \log(n))$

```
1: procedure NaiveVoronoi(P)
2:     V ← ∅
3:     for i ← 1, ..., n do                          ▷ O(n² log(n))
4:         V_p ← ℝ²
5:         for j ← 1, ..., n do                      ▷ O(n log(n))
6:             if i ≠ j then
```

```
7:                 V_p ← V_p ∩ H_ij
8:             end if
9:         end for
10:        V.insert(V_p)
11:    end for
12:    return V
13: end procedure
```

As can be easily seen, the runtime of this algorithm is $\mathcal{O}(n^2 \log(n))$ although $\mathrm{Vor}(P)$ only needs $\mathcal{O}(n)$ memory. This raises the question for the existence of a more efficient algorithm.

10.4.2 Fortune's algorithm

We start this section with the following observation:

Proposition 10.6

A point p in the plane is a Voronoi vertex if and only if the largest empty circle $c_P(p)$, i.e. the largest circle containing no points in its besides p with p being its midpoint, contains at least three points of P on its boundary. Moreover, a bisector m_{ij} defines a Voronoi edge if and only if there is a point $p \in m_{ij}$, whose largest empty circle $c_P(p)$ contains exactly two points on its boundary.

Proof. For both statements we show both implications.

'$\Leftarrow$': Let p be a point with empty circle $c_P(p)$ through points q_i, q_j and q_k. Then, at least the three cells $V(q_i), V(q_j)$ and $V(q_k)$ meet in p since they have the same distance from it, thus p must be a voronoi vertex.

'$\Rightarrow$': Conversely, if p is a Voronoi vertex, then there are at least three cells containing p on their boundary. Thus, there must be at least three points q_i, q_j and q_k on the boundary of $c_P(p)$.

For the second statement we get the following.

'$\Leftarrow$': Let p be a point with empty circle $c_P(p)$ through two points q_i and q_j. Then p belongs to the edge shared between $V(q_i)$ and $V(q_j)$. By the previous statement p cannot be a Voronoi vertex and thus has to be a Voronoi edge.

'$\Rightarrow$': Finally let m_{ij} contain an edge $e \in \mathrm{Vor}(P)$. Then for every point $p \in e$, the largest empty circle $c_P(p)$ contains only the points q_i and q_j defining the adjacent cells on its boundary.

This shows both statements. □

Using this characterization, we will employ a sweep line ℓ scanning the points in P from top to bottom. This sweep line should inhibit the property that points below it can still change the Voronoi diagram in a region below a curve β above ℓ. We will call this curve the beach-line. Since the beach-line separates the region that can still change from the one that is already computed correctly[2], we will maintain it in a status queue.

[2]The beach-line should be defined in a way such that the Voronoi vertices and edges arise in a natural way.

To implement this algorithm we still need to derive a representation of the curve β. To do this, we note that for the mentioned regions to be separated, the points on β must satisfy

$$\beta(x) - \ell_y = \min_{p \in P} \|p - (x, \beta(x))^T\|_2,$$

i.e. the beach-line consists of all points that have the same distance to the sweep line as to the closest point $p \in P$ above ℓ. Solving this equation[3] yields

$$\beta(x) = \min_{p \in P} \frac{x^2 - 2p_x x + p_x^2 + p_y^2 - \ell_y}{2(p_y - \ell_y)}.$$

Thus β is a piecewise quadratic curve, i.e. the beach-line β consists out of multiple parabolic arcs β_i.
Now that we know how to separate those regions, we need to figure out how the structure of the Voronoi diagram arises above β. Again, we use the fact that the Voronoi diagram, at least in our case of using the Euclidean metric, consists only out of half-lines and line segments. Hence, we only need to track the arising vertices and their connections as the sweep line and thus the beach-line traverses the plane. To do so, we will introduce two events; a site event and a circle event. A site, respectively, circle event occurs when a new arc appears, respectively, disappears in the beach-line.

Lemma 10.7

Only a site event can contribute to a new arc in the beach-line.

Proof. Assume the contrary. Then, an arc could appear by another parabola breaking through the beach-line from above. This could happen in two ways:

1. Assume an arc β_j breaks through the middle of another arc β_i from above. Then there is a point in time t_0 where $\ell_y < p_y^{(i)}$, $\ell_y < p_y^{(j)}$ and β_i and β_j touch in a single point q, where $p^{(i)}$ and $p^{(j)}$ are the points defining the arcs β_i and β_j. Thus, before t_0 the arcs β_i and β_j cannot intersect in any point. Now consider the leading coefficients of both arcs

$$c(\beta_i) = \frac{1}{2(p_y^{(i)} - \ell_y)} \quad \text{and} \quad c(\beta_j) = \frac{1}{2(p_y^{(j)} - \ell_y)}.$$

   We observe that the arcs get wider, the lower the sweep line moves below the point they were created at. Since β_j breaks through β_i, $p^{(j)}$ must have been passed by ℓ before $p^{(i)}$. Thus, at time t_0, we must have $c(\beta_i) \geq c(\beta_j)$, implying that β_j is wider than β_i. This contradicts the fact that they have not intersected before time t_0.

2. Now assume that an arc β_j appears at the transition point q of two arcs β_i and β_k. Then there is a circle C through $p^{(i)}, p^{(j)}$ and $p^{(k)}$, where $p^{(\cdot)}$ is defined as above, which touches the sweep line at a certain time t_0. After t_0 the circle C, still tangential to ℓ and passing through $p^{(j)}$ and one of the other points, must have either swallowed $p^{(i)}$ or $p^{(k)}$, i.e. this point now lies in the interior of C. We may assume the swallowed point is $p^{(k)}$. Since $p^{(k)}$ is inside of C it must be closer to q than $p^{(j)}$. Thus, by the properties of the beach-line β_j cannot contribute to the beach-line immediately after t_0, which is a contradiction to it passing the other arcs after t_0.

Since neither case can occur, the statement must follow. □

[3] This is the point where the choice of the Euclidean metric is crucial for us.

Remark: It is easy to see that beach-line consists of at most $2n-1$ parabolic arcs, since every site event adds one more arc that can split arcs that are already part of the beach line into two new arcs.

Lemma 10.8

An arc can only disappear from the beach-line at a circle event.

Proof. The only other way of an arc disappearing is if it gets overtaken by another arc from above. This cannot happen as we already have shown. □

Lemma 10.9

All Voronoi vertices can be detected by the circle events.

Proof. We have to show that the points p_i, p_j and p_k of adjacent Voronoi cells have generated adjacent arcs before the corresponding circle event would have to be processed. Lifting the sweep line by $\varepsilon > 0$ gives two empty circles tangential to ℓ interpolating p_i and p_j and p_j and p_k respectively. Thus, prior to the to be processed circle event the arcs β_i and β_j and also β_j and β_k were adjacent, hence they created the circle event before it needed to be handled. □

The above lemmas show that the circle events can be calculated during the sweeping process. Thus, our algorithm only needs the site events, i.e. the set of points P, as input. Moreover, it can happen that a site event prevents a previously identified circle event which then again must be removed from the event queue.
Before continuing, we want to summarize the results we have obtained so far in an instructive manner. To implement the algorithm as motivated by the results above we need:

1. A priority queue Q for site and circle events, where the priority is the y-coordinate of the eventpoint. This queue will be initialized with the site events induced by P.

2. A self-balancing search tree T representing the structure of the beach-line β. This allows querying for arcs involved int the current event in $\mathcal{O}(n \log(n))$ time.
   The leaves γ of T will represent the different arcs of the beach line from left to right and contain a pointer to the point the arc was generated from, a pointer to a circle event removing the arc if one exists and pointers to the left and right arc in the beach-line. The inner nodes $[p_i, p_j]$ on the other hand, represent the transition points q_{ij} between arcs β_i and β_j and contain pointers to the arc-generating points p_i and p_j and a pointer to the corresponding Voronoi edge in D.

3. A DCEL D representing the Voronoi diagram. Again, to avoid half-lines we embed P in a sufficiently large bounding box R.

With this setup we can now describe the algorithm originally published by Steven Fortune in 1986 in detail:

Algorithm 18 Fortune's algorithm

Input: A set $P = \{p_1, \ldots, p_n\}$ of n distinct points in the plane.
Output: The voronoi diagram $\mathrm{Vor}(P)$ of P as a DCEL with a bounding box R.
Runtime: $\mathcal{O}(n \log(n))$

```
1: procedure VoronoiDiagram(P)
2:     R ← ComputeBoundingBox(P)
3:     Q ← InitializeEventQueue(P)                                   ▷ O(n log(n))
4:     T ← ∅
5:     D ← ∅
6:     while Q ≠ ∅ do                          ▷ O(n log(n)) = O((n + 2n − 5) log(n))
7:         e ← Q.pop()
8:         if e.type = SITE then
9:             HandleSiteEvent(e.p, T, D)                            ▷ O(log(n))
10:        else
11:            HandleCircleEvent(e.involved, e.midpoint, T, D)       ▷ O(log(n))
12:        end if
13:    end while
14:    H ← T.getHalfLines()
15:    D.addBoundingBox(H, R)
16:    return D
17: end procedure
18: procedure HandleSiteEvent(p, T, D)
19:    if T.isEmpty() then
20:        T.insert(p)
21:    else
22:        γ ← T.findArcNodeAbove(p)
23:        if γ.hasCircleEvent() then
24:            Q.remove(γ.circleEvent)
25:        end if
26:        T.replace(γ, (γ.p, p, γ.p), ([γ.p, p], [p, γ.p]))
27:        D.addEdge(D.getCell(p), D.getCell(γ.p))
28:        if CreateCircleEvent(γ.arc, γ.arc.right, γ.arc.right.right) then
29:            e ← NewCircleEvent(γ.arc, γ.arc.right, γ.arc.right.right)
30:            Q.insert(e)
31:            T.updatePointers(γ.arc, γ.arc.right, γ.arc.right.right, e)
32:        end if
33:        if CreateCircleEvent(γ.arc, γ.arc.left, γ.arc.left.left) then
34:            e ← NewCircleEvent(γ.arc, γ.arc.left, γ.arc.left.left)
35:            Q.insert(e)
36:            T.updatePointers(γ.arc, γ.arc.left, γ.arc.left.left, e)
37:        end if
38:    end if
39: end procedure
40: procedure HandleCircleEvent(β, p, T, D)
41:    β_L, β_M, β_R ← β
42:    for e ∈ Q do
43:        if e.type = CIRCLE ∧ β_M ∈ e.involved then
44:            Q.remove(e)
45:        end if
46:    end for
47:    T.remove(β_M)
48:    T.updateInnerNodes()
49:    D.addVertex(p)
50:    D.addEdge(D.getCell(β_L.p), D.getCell(β_R.p))
```

```
51:     for i = 0, ..., |T.leaves| do
52:         γ ← T.leaves[i]
53:         if CreateCircleEvent(γ.arc.left.γ.arc, γ.arc.right) then
54:             e ← NewCircleEvent(γ.arc.left.γ.arc, γ.arc.right)
55:             Q.insert(e)
56:             T.updatePointers(γ.arc.left.γ.arc, γ.arc.right, e)
57:         end if
58:     end for
59: end procedure
```

Remark: In CreateCircleEvent, one must check if the transition points move towards each other. Since this is a simple line intersection check, the y-coordiante of these converging edges can be computed efficiently.

Theorem 10.10

The algorithm proposed by Fortune runs in $\mathcal{O}(n \log(n))$ time and uses $\mathcal{O}(n)$ memory.

Proof. Since the operations on Q, T and D take only $\mathcal{O}(\log(n))$ time, any event can be processed in $\mathcal{O}(\log(n))$ time. Moreover, since the total number of arcs is bounded by $2n - 1$, the memory requirements for T and Q are in $\mathcal{O}(n)$. □

Before ending this chapter, we will list some edge cases that still need to be considered:

- ▷ If two points have the same y-coordinate, any sequence of processing them is admissible. If those happen to be the first two points in P, then there will be no arc above the second point.
- ▷ If four or more points lie on one circle, then two or more circle events coincide. This can be handled by adding an edge of length zero to D which can be removed in a post-processing step.
- ▷ If a site event occurs at the transition of two arcs, one of the neighbors is split to yield an arc of length zero. This arc then will generate a circle event removing that arc as soon as its being processed.

Chapter 11

Delaunay Triangulation

11.1 Definitions

In a previous chapter we have used the concept of triangulations to solve the so called art gallery problem (cf. definition 5.1 in chapter 5). Back then, we did not rigorously define the term triangulation, since we only needed it as a tool to solve the art gallery problem and thus used the term triangulation as a purely geometric one. In this chapter however, we will define the term triangulation and its variants in more detail.

Definition 11.1 (Maximal planar subdivision)

A *maximal planar subdivision* is a subdivision S such that no edge connecting two vertices in S can be added to S without destroying its planarity.

Definition 11.2 (Triangulation)

Let $P = \{p_1, \ldots, p_n\}$ be a set of n points in the plane. Then, a *triangulation* of P is a maximal planar subdivision whose vertex set coincides with P.

As in the case with convex hulls in chapter 2, triangulations do not need to be unique. However, there are some triangulations that are more desirable than others. For example, we want to avoid triangles with obtuse angles, since they are often prone to numerical errors. Thus, the following definitions come into play.

Definition 11.3 (Angle-optimal triangulation)

Let T be a triangulation of a point set P and suppose it has m triangles. Then...

(i) the angle-vector of T is defined as $\alpha(T) := (\alpha_1, \ldots, \alpha_{3m})$, where α_i is the $(3m-i+1)$-th largest angle.

(ii) $\alpha(T)$ is lexicographically larger than $\alpha(T')$ if and only if $\alpha_i > \alpha'_i$ for some i and $\alpha_j = \alpha'_j$ for all $j < i$.

(iii) a triangulation T is called *angle-optimal* if $\alpha(T) \geq \alpha(T')$ for all triangulations T' of P.

Definition 11.4 (Delaunay triangulation)

A *Delaunay triangulation* is a triangulation T maximizing $\min \alpha(T)$.

At last, we will fix some notation for concepts that we will use at a later point in this chapter.

Definition 11.5 (Delaunay graph)

Let P be a set of n distinct points in the plane that do not lie on a common line. Then, the Delaunay graph[1] $\mathrm{DG}(P) = (P, E)$ is the dual graph to the Voronoi diagram induced by P.

Definition 11.6 (Illegal edges)

Consider the (only) two triangulations[2] T and T' of a convex quadrilateral $p_1p_2p_3p_4$. Then, the diagonal edge $\overline{p_1p_3}$ in T is called *illegal*, if $\min \alpha(T) < \min \alpha(T')$. Thus, an edge within a convex quadrilateral is illegal, if flipping it increases $\alpha(T)$.

Definition 11.7 (Legal triangulation)

A triangulation T is *legal* if it does not contain any illegal edges, i.e. edges that would increase $\alpha(T)$ when flipping them.

Although they might seem quite arbitrary right now, they will prove themselves to be of great use later on.

11.2 Applications

With our notations fixed, we can now turn to the applications of triangulations. In the following, we will discuss two applications of triangulations in computer graphics and engineering.

1. **Surfaces in computer graphics.**
   Problem: Given a set of points P on the surface of a virtual landscape, find a triangulation of P such that the triangles are as equilateral as possible to avoid rendering artifacts.
   Solution: Compute a angle-optimal triangulation of P.

2. **Finite elements method.**
   Problem: Given a set of collocation points P, generate a triangulation suitable for the finite element method.
   Solution: Since the quality of finite element solvers depends on the maximal obtuseness of the triangulation chosen, we want to generate an angle-optimal triangulation of P.

[1] Since the faces are not necessarily triangles, we call this object Delaunay graph instead of Delaunay triangulation

[2] These triangulations only differ in one internal, namely the diagonal, edge. Thus, they can be transformed into each other by flipping said edge.

11.3 Analysis

In this section, we will prove some basic properties of triangulations. We will start with a lemma that will give us an important identity involving the number of triangles and edges required for a triangulation, namely:

Lemma 11.8

Let P be a set of n points in the plane that do not lie on a common line and let $k := |\partial\text{conv}(P) \cap P|$. Then any triangulation of P has $2n-2-k$ triangles and $3n-3-k$ edges.

Proof. Let T be a triangulation of P with m triangles. Then $n_f = m + 1$, since we have to account for the unbounded face $f_\infty := \mathbb{R}^2 \setminus \text{conv}(P)$. Moreover, we have three edges for each triangle and k edges for f_∞. Since each edges is shared between two faces we get $n_e = \frac{3m+k}{2}$. Now apply Euler's formula with $n_v = n$ to obtain $m = 2n - 2 - k$ and $n_e = 3n - 3 - k$. □

Next, we want to break down the problem of finding an angle-optimal triangulation into smaller subproblems. To this end, we consider the most basic polygon which does not have a trivial triangulation; the quadrilateral.

Theorem 11.9 (Detection of illegal edges)

Let $p_1p_2p_3p_4$ be a convex equilateral and C be the circle through p_1, p_2 and p_3. Then, $\overline{p_1p_3}$ is illegal if and only if p_4 lies in the interior of C[3]. Moreover, if p_4 does not lie on the boundary of C, then exactly one of the diagonals in the quadrilateral is an illegal edge.

Proceeding with our analysis, we want to prove some properties involving the Delaunay graph indued by our point set P.

Lemma 11.10

The Delaunay graph $\text{DG}(P)$ of a point set P is a planar graph.

Proof. We only give an idea to this proof: Use contradiction and argue by considering the largest empty circles in the Voronoi diagram and their intersections with edges. □

Since we are working with planar graphs and empty circles, we can restate a result we derived in the context of Voronoi diagrams:

Lemma 11.11

Let P be a set of points in the plane. Then, three points p_i, p_j and p_k in P are vertices of the same face in $\text{DG}(P)$ if and only if the circle containing p_i, p_j and p_k contains no point of P in its interior.

This now immediately implies the following result:

[3]This result allows to efficiently check whether a given edge e is legal or not.

Theorem 11.12 (Characterization of Delaunay triangulations)

Let P be a set of points in the plane and let T be a triangulation of P. Then, T is a Delaunay triangulation of P if and only if the circumcircle of any triangle in T contains no point of P in its interior.

One theorem that will be of great use in the following is a generalization of Thales' theorem. In most cases, Thales' theorem is stated like this:

> *If A, B, and C are distinct points on a circle where the line $\overline{AC}$ is a diameter, the angle $\measuredangle ABC$ is a right angle.*

We however deal with lines that are not necessarily diameters of the circle in question. Thus, we need a generalized result called the inscribed angle theorem.

Theorem 11.13 (Inscribed angle theorem)

Let C be a circle, ℓ a line passing through C with $\ell \cap C = \{a, b\}$ and p, q, r and s points lying on the same side of ℓ with p and q lying on, r inside of and s outside of C. Then

$$\measuredangle arb > \measuredangle apb = \measuredangle aqb > \measuredangle asb.$$

With this theorem, we are now able to prove the following lemma.

Lemma 11.14

Let P be a set of points in the plane. Then, a triangulation T of P is legal if and only if T is a Delaunay triangulation of P.

Proof. Again, we only give an idea to this proof:

> '$\Rightarrow$': Assume the contrary. Then there exists a triangle that contains a point of P within its circumcircle. By the definition of a legal edge, the result about detecting illegal edges and the inscribed angle theorem, we obtain the contradiction.
>
> '$\Leftarrow$': By definition.

This then shows the claim. □

We are now ready to prove the main result of this section:

Theorem 11.15 (Optimality of Delaunay triangulations)

Let P be a set of points in the plane. Then any angle-optimal triangulation of P is a Delaunay triangulation of P. Moreover, any Delaunay triangulation of P maximizes the minimum angle across all triangulations of P.

Proof. Since any angle-optimal triangulation T must be legal, it follows by the previous lemma that T is also a Delaunay triangulation of P. In particular, if there are no more than three points on any empty circle's boundary, then there exists only one legal triangulation, i.e. the angle-optimal, hence Delaunay triangulation of P is unique. □

11.4 Algorithms

Now that we have analyzed the problem at hand in great detail, we want to put our focus on developing efficient algorithms for the problem. As always, we start by specifying what we want to achieve:

> Given a set of points P, compute a triangulation of P that maximizes the minimum angle across all triangulations of P.

We will start by presenting a naive iterative algorithm.

11.4.1 Naive incremental algorithm

Since flipping an illegal edge in a triangulation T increases $\alpha(T)$, iteratively flipping all illegal edges yields an optimal triangulation.

Algorithm 19 Legal triangulation

Input: A triangulation T of a set of points P.
Output: A legal triangulation of P
Runtime: $\mathcal{O}(n^2)$

```
1: procedure LegalTriangulation(T)
2:     while T.illegalEdges ≠ ∅ do
3:         e ← T.illegalEdges[0]
4:         △apb ← e.incidentFace
5:         △aqb ← e.twin.incidentFace
6:         T.remove(e)
7:         T.insert(pq)
8:     end while
9:     return T
10: end procedure
```

Since every flip strictly improves the triangulation and there are only finitely different triangulations, the algorithm must terminate.

11.4.2 Voronoi based triangulation

Another, more efficient approach, would be to use the theorems stated above to proceed as follows:

Outline (VoronoiTriangulation):

1. Compute the Voronoi diagram of P.
2. Compute its dual graph DG(P).

This approach however requires an available implementation of the Voronoi diagram algorithm which might not be given. Moreover, if the points in P happen to lie on a common circle and $|P| \geq 4$, the Delaunay graph really is a graph and not a triangulation. Thus, we need to find a robust alternative approach independent of the Voronoi diagram algorithm.

11.4.3 Fast incremental algorithm

Like in the case with the point location problem in chapter 9, we start out by finding a single triangle or a quadrilateral bounding box made out of two triangles that contain the entirety of P and make up a legal triangulation of the quadrilateral. Then we add the points of P in a random order and create an appropriate amount of new triangles each step. To obtain a legal triangulation we also restore the Delaunay property, i.e. the non-existence of illegal edges, after each insertion. If more than three points lie on the same circle, the corresponding face can be triangulated arbitrarily[4].

Algorithm 20 Delaunay triangulation

Input: A set of n distinct points $P = \{p_1, \ldots, p_n\}$ in the plane.
Output: A Delaunay triangulation T of P.
Runtime: $\mathcal{O}^*(n \log(n))$

```
 1: procedure DelaunayTriangulation(P)
 2:     P ← Shuffle(P)
 3:     T ← BoundingTriangle(P)
 4:     for r = 1, ..., n do
 5:         T^(1), T^(2) ← T.query(p_r)
 6:         if T^(2) = None then
 7:             △p_i p_j p_k ← T^(1)
 8:             T.remove(T^(1))
 9:             T.insert(△p_i p_r p_k, △p_i p_j p_r, △p_j p_k p_r)
10:             for e = p_i p_j, p_j p_k, p_k p_i do
11:                 LegalizeEdge(p_r, e, T)
12:             end for
13:         else
14:             △p_i p_k p_j ← T^(1)
15:             △p_i p_l p_j ← T^(2)
16:             T.remove(T^(1), T^(2))
17:             T.insert(△p_i p_r p_l, △p_l p_j p_r, △p_j p_r p_k, △p_k p_r p_i)
18:             for e = p_i p_l, p_l p_j, p_j p_k, p_k p_i do
19:                 LegalizeEdge(p_r, e, T)
20:             end for
21:         end if
22:     end for
23:     T.removeBoundingTriangles()
24:     return T
25: end procedure
26: procedure LegalizeEdge(p_r, p_i p_j, T)
27:     if IsIllegal(p_i p_j) then
28:         △p_i p_j p_k ← △p_r p_i p_j.adjacentAlong(p_i p_j)
29:         T.remove(p_i p_j)
30:         T.insert(p_r p_k)
31:         LegalizeEdge(p_r, p_i p_k, T)
32:         LegalizeEdge(p_r, p_k p_j, T)
33:     end if
34: end procedure
```

[4] In this case the Delaunay triangulation is not uniquely determined.

Remark: To avoid cluttered notation, we updated the set P in the beginning by shuffling it in-place instead of creating a new set P^* like we did in the point location algorithm.

Theorem 11.16

The algorithm DelaunayTriangulation correctly computes a Delaunay triangulation of P in $\mathcal{O}(n \log(n))$ expected time.

Proof. First, we show that all new triangles satisfy the Delaunay property at the end of each step.
Assume one of the triangles T_r generated during the insertion of a point p_r does not satisfy the Delaunay condition. Then, an edge between p_r and the triangle Δ that was generated by removing the triangles must be flipped. This yields a new triangle Γ of consecutive points p_i, p_j and p_k of Δ. If $\Gamma \in T$ before the insertion, Γ was removed, since it violated the Delaunay condition with p_r. If, on the other hand, $\Gamma \notin T$ before the insertion occurred, it did not satisfy the Delaunay condition even before p_r was inserted.
Since both cases yield a contradiction the algorithm must yield a correct Delaunay triangulation.
Now to the second statement. Naively implementing this algorithm still yields a runtime of $\mathcal{O}(n^2)$ because the search for a single triangle containing a point p_r takes $\mathcal{O}(r)$ time. Using a trapezoidal map on the other hand reduces the expected runtime for queries to $\mathcal{O}^*(\log(n))$. Since the expected number of triangles that must be removed ins constant in every step, we get a total expected runtime of $\mathcal{O}^*(n \log(n))$ for the algorithm. □

Remark: The lowest bound for the runtime of the Delaunay and Voronoi construction is in $\Omega(n \log(n))$. Nevertheless, the memory requirements are still in $\mathcal{O}(n)$.

Chapter 12

Recap

In the final chapter of these lecture notes we want to give a brief overview of the algorithms we have seen in this lecture. It is meant as a quick reference for the reader, and as a summary of the algorithms we have seen. We will not go into detail here, but rather give a short description of the algorithms and their expected runtime and memory usage.
Besides that we will reiterate on the most important concepts and how they interplayed with each other across the multiple chapters.

12.1 Algorithm summary

Problem class	Approach	'Best' algorithm	(Expected) runtime	/ memory
Convex hull	Sweep line (radial)	Graham's scan	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
	Sweep line (radial)	Jarvis' march	$\mathcal{O}(nh)$	$\mathcal{O}(n)$
Line segment intersection	Sweep line	Bentley-Ottmann	$\mathcal{O}((n + \lvert I \rvert) \log(n))$	$\mathcal{O}(n)$
Map overlay	Sweep line	MapOverlay	$\mathcal{O}((n + k) \log(n))$	$\mathcal{O}(n)$
Polygon triangulation	Sweep line	MonotonePartition	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
		TriangulateMonotone	$\mathcal{O}(n)$	
		Triangulate	$\mathcal{O}(n \log(n))$	
Quadtrees	Spatial subdivision	Construction	$\mathcal{O}((d+1)n)$	$\mathcal{O}((d+1)n)$
		Query	$\mathcal{O}(d+1)$	
		NeighborSearch	$\mathcal{O}(d+1)$	
kD-trees	Spatial subdivision	BuildkDTree	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
		RangeSearch	$\mathcal{O}(\lvert T \cap R \rvert + n^{1-\frac{1}{k}})$	
Range-trees	Spatial subdivision	Construction	$\mathcal{O}(n \log(n)^{k-1})$	$\mathcal{O}(n \log(n)^{k-1})$
		RangeSearch	$\mathcal{O}(\lvert T \cap R \rvert + \log(n)^k)$	
BSP trees	Spatial subdivision	BSP2D (with AP)	$\mathcal{O}^*(n^2 \log(n))$	$\mathcal{O}(n \log(n))$
		BSP2D (without AP)	$\mathcal{O}(n \log(n))$	
Point location	Spatial subdivision	TrapezoidalMap	$\mathcal{O}^*(n \log(n))$	$\mathcal{O}^*(n)$
		PointQuery	$\mathcal{O}^*(\log(n))$	
Voronoi diagrams	Sweep line	Fortune's algorithm	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
Delaunay triangulation	Randomized	DelaunayTriangulation	$\mathcal{O}^*(n \log(n))$	$\mathcal{O}(n)$

Table 12.1: A summary of all important algorithms with their expected runtime and memory usage. For the variables influencing the runtime, please refer to the respective sections, as th same symbol may denote different quantities across chapters.

12.2 Sweep line algorithms

- **Requirements:** An event queue Q for storing the points where the sweep line has to do something. Then, a status queue T dynamically handling structural changes relevant to the problem. Finally an output data structure D representing the results.

- ▷ **Proof of correctness:** By induction over an invariant induced by the sweep line.
- ▷ **Applications:** Convex hull, line segment intersection, map overlay, polygon triangulation and Voronoi diagrams.

12.3 General concepts

- ▷ **Proves:** Deriving running times and memory requirements. Providing lower bounds proved by reduction. Applying Euler's formula for planar graphs.
- ▷ **General position:** Special arrangement of points and lines for edge-case avoidance. Can be implemented symbolically with lexicographic ordering.
- ▷ **Output-sensitive algorithms:** Some algorithms have a runtime depending on the output they generate. For example Jarvis' march or RangeSearch.
- ▷ **Dual graphs:** Dual graphs are used to translate the view from a vertex-edge correspondence to a edge-face correspondence. This was used for the art gallery problem or the Voronoi diagram approach for computing a Delaunay triangulation.

12.4 Data structures

- ▷ **DCELs:** Doubly connected edge lists were very useful for handling mesh manipulations.
- ▷ **Spatial subdivisions:** To find things in space, it is usually useful to strategically search it. This approach was used in the construction of quad trees, kD- and range trees, BSP trees, trapezoidal maps and Voronoi diagrams.
- ▷ **Triangulation:** To be able to describe objects of any shape or form we used triangulations to create meshes that consist of the most simple parts, i.e. triangles.

12.5 Randomized algorithms

- ▷ **Characteristics:** Random algorithms usually define themselves by non-deterministic behavior caused by either a shuffled input or a random selection of the next element considered in the algorithm.
- ▷ **Expected runtime:** Since the algorithm behave different each time they are executed, a new metric to measure their performance is introduced, namely the expected runtime. To calculate this runtime, one usually defines indicator variables carrying enough information about the temporal behavior they encompass and summing up their expected outcomes to then fall back on a comparison with the harmonic series to obtain a logarithmic upper bound.
- ▷ **Applications:** Random algorithms were used to improve the performance in constructing BSP trees, trapezoidal maps and Delaunay triangulations.