An Optimal-Time Algorithm for Shortest Paths on a Convex Polytope in Three Dimensions*

Yevgeny Schreiber[†] Micha Sharir[‡]

July 12, 2007

Abstract

We present an optimal-time algorithm for computing (an implicit representation of) the shortest-path map from a fixed source s on the surface of a convex polytope P in three dimensions. Our algorithm runs in $O(n \log n)$ time and requires $O(n \log n)$ space, where n is the number of edges of P. The algorithm is based on the $O(n \log n)$ algorithm of Hershberger and Suri for shortest paths in the plane [22], and similarly follows the continuous Dijkstra paradigm, which propagates a "wavefront" from s along ∂P. This is effected by generalizing the concept of conforming subdivision of the free space used in [22], and by adapting it for the case of a convex polytope in $\mathbb{R}^3$, allowing the algorithm to accomplish the propagation in discrete steps, between the "transparent" edges of the subdivision. The algorithm constructs a dynamic version of Mount's data structure [32] that implicitly encodes the shortest paths from s to all other points of the surface. This structure allows us to answer single-source shortest-path queries, where the length of the path, as well as its combinatorial type, can be reported in $O(\log n)$ time; the actual path can be reported in additional $O(k)$ time, where k is the number of polytope edges crossed by the path.

The algorithm generalizes to the case of m source points to yield an implicit representation of the geodesic Voronoi diagram of m sites on the surface of P, in time $O((n+m)\log(n+m))$, so that the site closest to a query point can be reported in time $O(\log(n+m))$.

Although several key ingredients of the algorithm are adapted from [22], this adaptation is quite challenging, and its implementation requires several additional nontrivial techniques.

*Work on this paper was supported by NSF Grants CCR-00-98246 and CCF-05-14079, by a grant from the U.S.-Israeli Binational Science Foundation, by grant 155/05 from the Israel Science Fund, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University. The paper is based on the Ph.D. Thesis of the first author, supervised by the second author. A preliminary version has been presented in Proc. 22nd Annu. ACM Sympos. Comput. Geom., pp. 30–39, 2006.

[†]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. syevgeny@tau.ac.il.

[‡]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA. michas@tau.ac.il.

1 Introduction

1.1 Background

The problem of determining the Euclidean shortest path on the surface of a convex polytope in $\mathbb{R}^3$ between two points is a classical problem in geometric optimization, which is motivated by many applications, such as robotics, terrain navigation, and industrial automation. This problem is a special case of the following basic general problem: Given a collection of obstacles (of known shapes and locations), find a Euclidean shortest obstacle-avoiding path between two given points, or, more generally, compute a compact representation of all such paths that emanate from a fixed source point. A much broader collection of specific problems, under various assumptions involving the dimension, metric, shape of the obstacles, and additional constraints on the path, are discussed in the survey of Mitchell [29]; here we mention only the results that are most relevant to our specific problem.

The first paper in computational geometry that has studied the single source shortest path problem on a single convex polytope P in $\mathbb{R}^3$ is by Sharir and Schorr [40]. Their algorithm runs in $O(n^3 \log n)$ time, where n is the number of vertices (or, in view of Euler's polyhedral formula, the number of edges or facets) of P. The algorithm constructs a planar layout of the *shortest path map*, and then the length and combinatorial type of the shortest path from the fixed source point s to any given query point q can be found, using point location, in $O(\log n)$ time; the path itself can be reported in $O(k)$ additional time, where k is the number of edges of the polytope that are traversed by the shortest path from s to q. Soon afterwards, Mount [31] gave an improved algorithm for convex polytopes with running time $O(n^2 \log n)$. Moreover, in [32], Mount has shown that the problem of storing shortest path information can be treated separately from the problem of computing it, presenting a data structure of $O(n \log n)$ space that supports $O(\log n)$-time shortest-path queries. However, the question whether this data structure can be constructed in subquadratic time, has been left open.

For a general, possibly nonconvex polyhedral surface, O'Rourke et al. [36] gave an $O(n^5)$-time algorithm for the single source shortest path problem. Subsequently, Mitchell et al. [30] presented an $O(n^2 \log n)$ algorithm, extending the technique of Mount [31]. All algorithms in [30, 31, 40] use the same general approach, called "continuous Dijkstra", first formalized in [30]. The technique keeps track of all the points on the surface whose shortest path distance to the source s has the same value t, and maintains this "wavefront" as t increases. It is easily seen that the wavefront consists of a collection of circular arcs, which can change combinatorially when one of the arcs hits a vertex or edge of P, when two different arcs run into each other, or when an arc shrinks to a point and is then eliminated by its two neighboring arcs. The continuous Dijkstra approach maintains a priority queue of future critical events where the wavefront undergoes such combinatorial changes, where the priority of an event is its shortest path distance from s. The approach treats certain elements of ∂P (vertices, edges, or other elements) as nodes in a graph, and follows Dijkstra's algorithm [13] to extract the unprocessed element currently closest to s and to propagate from it, in a continuous manner, shortest paths to other elements. The same general approach is also

used in our algorithm.

Chen and Han [10] use a rather different approach (for a not necessarily convex polyhedral surface). Their algorithm builds a shortest path sequence tree, using an observation that they call “one angle one split” to bound the number of branches, maintaining only $O(n)$ nodes in the tree in $O(n^2)$ total running time. The algorithm of [10] also constructs a planar layout of the shortest path map (which is “dual” to the layout of [40]), which can be used similarly for answering shortest path queries in $O(\log n)$ time (or $O(k + \log n)$ time for path reporting). (Their algorithm is somewhat simpler for the case of a convex polytope P, relying on the property, established by Aronov and O‘Rourke [7], that this layout of P does not overlap itself.) In [11], Chen and Han follow the general idea of Mount [32] to solve the problem of storing shortest path information separately, for a general, possibly nonconvex polyhedral surface. They obtain a tradeoff between query time complexity $O(d \log n/ \log d)$ and space complexity $O(n \log n/ \log d)$, where d is an adjustable parameter. Again, the question whether this data structure can be constructed in subquadratic time, has been left open.

The problem has been more or less “stuck” after Chen and Han’s paper, and the quadratic-time barrier seemed very difficult to break. For this and other reasons, several works [2, 3, 4, 5, 20, 21, 27, 28, 42] have presented approximate algorithms for the 3-dimensional shortest path problem. Nevertheless, the major problem of obtaining a subquadratic, or even near-linear, exact algorithm has remained open.

In 1999, Kapoor [24] has announced such an algorithm for the shortest path problem on an arbitrary polyhedral surface P (see also a review of the algorithm in O’Rourke’s column [33]). The algorithm follows the continuous Dijkstra paradigm, and claims to be able to compute a shortest path from the source s to a *single target point* t in $O(n \log^2 n)$ time (so it does not preprocess the surface for answering shortest path queries). Facets of P are unfolded into the plane of the face of s, and the main claim of the algorithm is that, using a set of complicated data structures (that represent convex hulls of pieces of the wavefront W, as well as convex hulls of pieces of the boundary B of the yet unexplored region that contains the target t, and the associations between the waves of W with their nearest neighbors in B), the total number of times that the data structures need to be updated in order to simulate the wavefront propagation is linear (and that each update can be performed in $O(\log^2 n)$ time, amortized).

However, as far as we know, the details of Kapoor’s algorithm have not yet been published, which makes it impossible to ascertain the correctness and the time complexity of the algorithm. Moreover, as it is presented, there seem to be several difficulties that remain to be solved in Kapoor’s approach. We list a few of these difficulties in Appendix B. As it is presented, we feel that the algorithm of Kapoor [24] has many issues to address and to fill in before it can be judged at all.

1.2 The algorithm of Hershberger and Suri for polygonal domains

A dramatic breakthrough on a loosely related problem has taken place in 1995,[1] when Hershberger and Suri [22] obtained an $O(n \log n)$-time algorithm for computing a shortest path between two points *in the plane* in the presence of polygonal obstacles (where n is the number of obstacle vertices). The algorithm actually computes a shortest path map from a fixed source point to all other (non-obstacle) points of the plane, which can be used to answer single-source shortest path queries in $O(\log n)$ time.

Since our algorithm uses (adapted variants of) many of the ingredients of Hershberger and Suri's algorithm, we provide a brief overview of their technique. The algorithm of [22] uses the continuous Dijkstra method — that is, propagation of the wavefront amid the obstacles, where each wave emanates from some obstacle vertex already covered by the wavefront. See Figure 1(a) for an illustration. During the wavefront propagation, critical events that change the wavefront topology are processed: wavefront-wavefront collisions, wavefront-obstacle collisions, and wave elimination within a single wavefront.

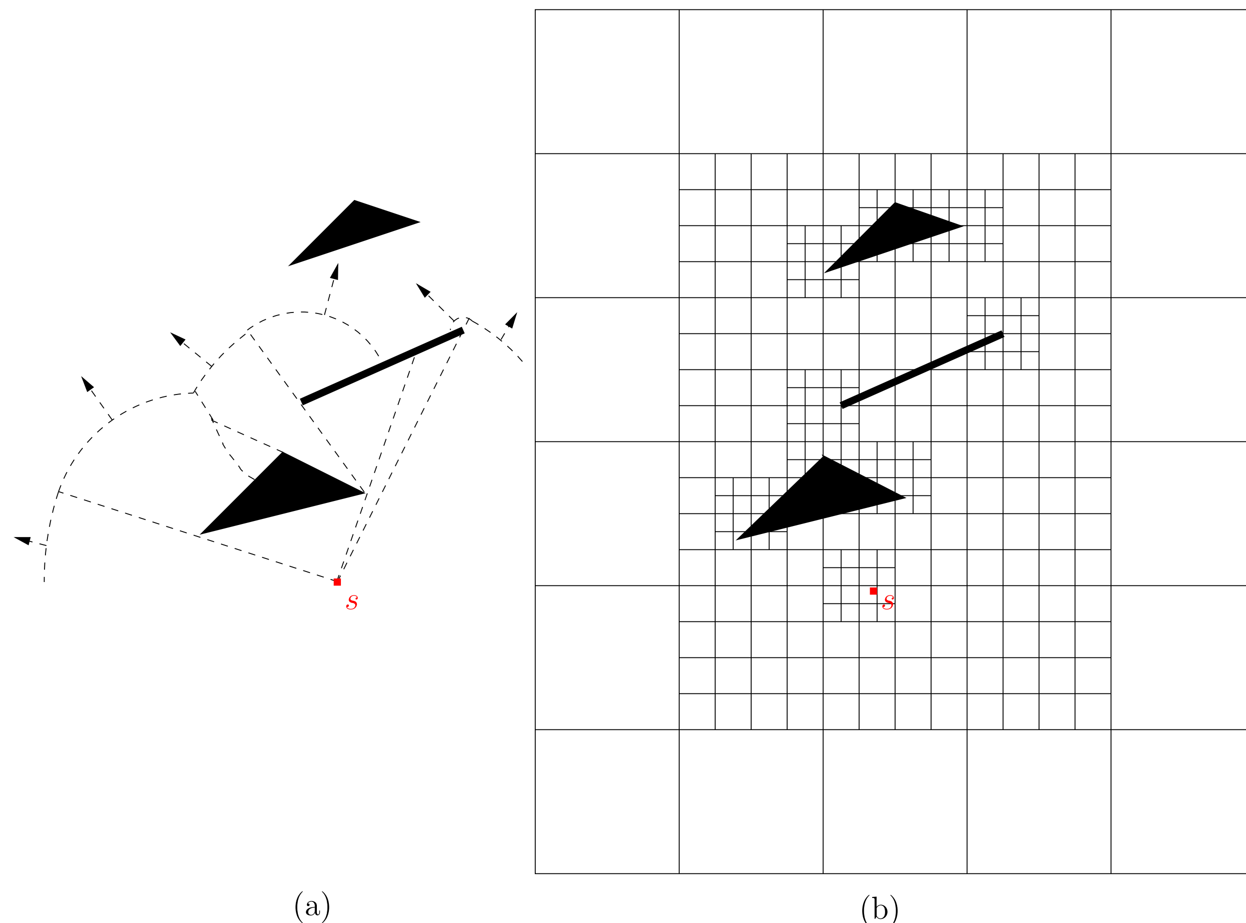

Figure 1: *The planar case: (a) The wavefront propagated from s, at some fixed time t. (b) The conforming subdivision of the free space.*

[1] A preliminary (symposium) version has appeared in 1993; the last version was published in 1999.

The key new ingredient in Hershberger and Suri's algorithm, which makes the wavefront propagation efficient, is a quad-tree-style subdivision of the plane, of size $O(n)$, on the vertices of the obstacles (temporarily ignoring the obstacle edges). See Figure 1(b) for an illustration. Each cell of this *conforming subdivision* is bounded by $O(1)$ axis-parallel straight line edges (called *transparent edges*), contains at most one obstacle vertex, and satisfies the following crucial "conforming" property: For any transparent edge e of the subdivision, there are only $O(1)$ cells within distance $2|e|$ of e. Then the obstacle edges are inserted into the subdivision, while maintaining both the linear size of the subdivision and its conforming property — except that now a transparent edge e has the property that there are $O(1)$ cells within *shortest path distance* $2|e|$ of e. These transparent edges form the elements on which the Dijkstra-style propagation is performed — at each step, the wavefront is ascertained to (completely) cover some transparent edge, and is then advanced into $O(1)$ nearby cells and edges. Since each cell has constant descriptive complexity, the wavefront propagation inside a cell can be implemented efficiently. The conforming nature of the subdivision guarantees the crucial property that each transparent edge e needs to be processed *only once*, in the sense that no path that reaches e after the simulation time at which it is processed can be a shortest path, so the Dijkstra style of propagation works correctly for the transparent edges.

During the propagation, the algorithm collects the wavefront collision data, from which the edges and vertices of the final map can be constructed. Inside a cell, a wavefront-obstacle collision event is relatively easy to handle; however, a wavefront-wavefront collision is more complex, especially when the colliding waves are not neighbors in the wavefront. The collision of neighboring waves occurs when a wave is eliminated by its two neighbors, which is easy to detect and process. To process collisions between non-neighboring waves another idea is introduced in [22] — the *approximate (or one-sided) wavefront*.

Propagating the exact wavefront that reaches a transparent edge e appears to be inefficient; instead, the algorithm maintains two separate "approximate" wavefronts approaching e from opposite sides. Together, this pair of one-sided wavefronts carry all the information needed to compute the exact wavefront at e. A limited interaction between this pair of wavefronts at e allows the algorithm to eliminate some of the superfluous waves and (implicitly) detect all wavefront-wavefront collisions (that constitute the vertices of the true shortest path map) when processing transparent edges that lie *in a small neighborhood* of the actual collision location. In other words, a superfluous wave that should have been eliminated in some cell may survive for a while, but it will travel through only $O(1)$ adjacent cells before being "caught" and destroyed, so the damage that it may have entailed till this point does not cause the asymptotic performance of the algorithm to deteriorate.

To track all the changes of the wavefront during the propagation, it is implemented as a persistent data structure that requires $O(\log n)$ space for each update, resulting in an algorithm with $O(n \log n)$ storage.

At the end of the propagation phase, all the collision information is collected, and then Voronoi diagram techniques are used to compute exactly the vertices of the shortest path map within each cell. The vertices in all the cells are then combined into a single map using standard plane sweeping and some additional tricks. Processing the resulting map for point location completes the algorithm.

1.3 An overview of our algorithm

Our algorithm follows the general outline of the technique of [22]: It constructs a conforming subdivision of the boundary ∂P of the given convex polytope P and applies the continuous Dijkstra propagation technique to the resulting transparent edges. However, extending the ideas of [22] to our case is quite involved, and requires special constructs, careful implementation, and finer analysis. In particular, many additional technical steps that address the 3-dimensional (non-flat) nature of the problem are introduced. To aid readers familiar with [22], the structure of our paper closely follows that of [22], although each part that corresponds to a part of [22] is quite different in technical details.

We begin with an overview of our algorithm. As in [22], we construct a conforming subdivision of ∂P to control the wavefront propagation. We first construct an oct-tree-like *3-dimensional* axis-parallel subdivision S_{3D}, only on the vertices of ∂P. Then we intersect S_{3D} with ∂P, to obtain a *conforming surface subdivision* S. (We use the term "facet" when referring to a triangle of ∂P, and we use the term "face" when referring to the square faces of the 3-dimensional cells of S_{3D}. Furthermore, each such face is subdivided into square "subfaces".) In our case, a transparent edge e may traverse many facets and edges of P, but we still want to treat it as a single simple entity. To this end, we first replace each actual intersection ξ of a subface of S_{3D} with ∂P by the *shortest path* on ∂P that connects the endpoints of ξ and traverses the same facet sequence of ∂P as ξ, and make those paths our transparent edges. (If these shortened paths cross each other, we split them into subedges at the crossing points.) We associate with each such transparent edge e the *polytope edge sequence* that it crosses, which is stored in compact form and is used to unfold e to a straight segment. To compute the unfolding efficiently, we preprocess ∂P into a *surface unfolding data structure*, that allows us to compute, in $O(\log n)$ time, the image of any query point $q \in \partial P$ in any unfolding formed by a contiguous sequence of polytope edges crossed by an *axis-parallel plane* that intersects the facet of q. This is a nontrivial addition to the machinery of [22]. (In contrast, in the planar case the transparent edges are simply straight segments, which are trivial to represent and to manipulate.)

Similarly to [22], we maintain a simulation timer to control the propagation of the wavefront from one transparent edge e of S to $O(1)$ transparent edges of nearby cells. Before doing so, we first consolidate the wavefronts that have already reached e, constructing a representation of the true wavefront at e at a time when e is ascertained to have been completely covered by the wavefront, but before the wavefront covers other transparent edges further from the source to which we want to propagate from e. (This representation uses one-sided wavefronts, as in [22] — see below.) The last transparent edges from which the contributing wavefronts were propagated to e bound the so-called *well-covering region* $R(e)$ of e, which has similar properties to those in [22]. A key difference is that in our case shortest paths "fold" over ∂P, and need to be unfolded onto some plane (on which they look like straight segments). We cannot afford to perform all these unfoldings explicitly — this would right away degrade the storage and running time to quadratic in the worst case. Instead we maintain partial unfolding transformations at the nodes of our structure, composing them on the fly (as rigid transformations of 3-space) to perform the actual unfoldings whenever needed (the same is done when unfolding the transparent edges themselves).

We maintain two *one-sided wavefronts* instead of one exact wavefront at each transparent edge e. We enforce the invariant that, for any point $p \in e$, the true shortest path distance from s to p is the smaller of the two distances to p encoded in the two one-sided wavefronts. Unlike [22], we do not apply any *explicit* interaction between the one-sided wavefronts. (However, there is still an implicit interaction between them, in the sense that a wave, reaching a transparent edge e later than e was ascertained to have been completely covered by an opposite one-sided wavefront, will not be propagated further.)

The need to unfold shortest paths onto a plane creates additional difficulties. On top of the main problem that a surface cell may intersect many (up to $\Omega(n)$) facets of P, it can in general be unfolded in more than one way, and such an unfolding may *overlap* itself (see [34, 45] for description of this problem).

To overcome this difficulty, we introduce a *Riemann structure* that efficiently represents the unfolded regions of the polytope surface that the algorithm processes. This representation subdivides each surface cell into $O(1)$ simple *building blocks* that have the property that a planar unfolding of such a block (a) is unique, and (b) is a simply connected polygon bounded by $O(1)$ straight line segments (and does not overlap itself). A global unfolding is a concatenation of unfolded images of a sequence, or more generally a tree, of certain blocks. It may overlap itself, but we ignore these overlaps, treating them as different layers of a Riemann surface. Each building block appears a constant number of times in the Riemann structure (of a fixed cell), and the overall structure has the property that it contains the shortest paths from the source to all the points of ∂P.

In summary, each step of the wavefront propagation phase picks up a transparent edge e, constructs each of the one-sided wavefronts at e by *merging* the wavefronts that have already reached e *from a fixed side*, and propagates from e each of its two one-sided wavefronts to $O(1)$ nearby transparent edges f, following the general scheme of [22]. Each propagation that reaches f from e proceeds along a fixed sequence of building blocks that connect e to f. Thus each propagation traces paths from a fixed *homotopy class* — they can be deformed into one another (inside the well-covering region where they are currently propagated), while continuing to trace the same edge and facet sequences of ∂P (as well as the same sequence of transparent edges). We call such a propagation *topologically constrained*, and denote the resulting wavefront that reaches f as $W(e, f)$, omitting for convenience the corresponding block sequence (or homotopy class). For a fixed edge e, there are only $O(1)$ successor transparent edges f and only $O(1)$ block sequences for any of those f's.

During each propagation, we keep track of combinatorial changes that occur *within* the wavefront, as it is being propagated from some predecessor edge g to e: At each of these events, we either split a wave into two waves when it hits a vertex, or eliminate a wave when it is "overtaken" by its two neighbors. Following a modified variant of the analysis of [22], we show that the algorithm encounters a total of only $O(n)$ "events", and processes each event in $O(\log n)$ time. To achieve the latter property, we represent each wavefront by a tree structure, as in [22], which supports standard tree operations (including SPLIT and CONCATENATE), priority queue operations (on the distances from each generator in W to the point where its wave is eliminated by its neighbors), and, a novelty of the structure, unfolding operations (that are constantly needed to trace and manipulate shortest paths as

unfolded straight segments). The collection of the "unfolding fields" in the resulting data structure is actually a dynamic version of the *incidence data structure* of Mount [32] that stores the incidence information between m nonintersecting geodesic paths and n polytope edges, and supports $O(\log(n+m))$-time shortest-path queries, using $O((n+m)\log(n+m))$ space. Our data structure has similar space requirements and query-time performance; the main novelty is the optimal preprocessing time of $O((n+m)\log(n+m))$ (Mount constructs his data structure in time proportional to the number of intersections between the polytope edges and the geodesic paths, which is $\Theta(nm)$ in the worst case). In this sense, we combine the benefits of the data structure of [22] with those of [32].

When all wavefronts, propagated from predecessor transparent edges, have reached e, we merge them into two one-sided wavefronts at e, similarly to the corresponding procedure in [22]. This happens at some simulation time t_e, which is an upper bound on the time at which e has been completely covered by the true wavefront. The main reason for maintaining one-sided wavefronts is that merging them is easy: As in [22], two such (topologically constrained) wavefronts $W(f,e)$, $W(g,e)$ *cannot interleave along* e, and each of them "claims" a contiguous portion of e (this property is false when merging wavefronts that reach e from different sides, or that are not topologically constrained). This allows us to perform the mergings in a total of $O(n\log n)$ time.

After the wavefront propagation phase, we perform further preprocessing to facilitate efficient processing of shortest path queries. This phase is rather different from the shortest path map construction in [22], since we do not provide, nor know how to construct, an explicit representation of the shortest path map on P in $o(n^2)$ time.[2] However, our implicit representation of all the shortest paths from the source suffices for answering any shortest path query in $O(\log n)$ time. Informally, we retrieve $O(1)$ candidates for the shortest path, and select the shortest among them. The query "identifies" the path combinatorially. It can produce right away the length of the path (assuming the real RAM model of computation), and the direction at which it leaves s to reach the query point. An explicit representation of the path takes $O(k)$ additional time, where k is the number of polytope edges crossed by the path.

The paper is organized as follows (again, we keep the paper structure as similar to [22] as possible). Section 2 provides some preliminary definitions and describes the construction of the conforming surface subdivision using an already constructed conforming 3D-subdivision S_{3D}, while the construction of S_{3D}, which is slightly more involved, is deferred to Section 6. While the construction of S_{3D} is very similar to the $2D$ construction given in [22], the construction in Section 2 is new and involves many ingredients that cater to the spatial structure of convex polytopes. Section 3 also has no parallel in [22]. It presents the Riemann structure and other constructs needed to unfold the polytope surface for the implementation of the wavefront propagation phase. Section 4 describes the wavefront propagation phase

[2] An explicit representation is tricky in any case, because the map, in its folded form, has quadratic complexity in the worst case. Our representation is actually an improved (dynamic) version of the compact implicit representation of [32], which, before the availability of our algorithm, was not known to be constructible in subquadratic time. There exist other compact implicit representations [10, 11] that allow various tradeoffs between the query time and the space complexity; however, so far (even with the availability of our algorithm) none of them is known to be constructible in subquadratic time.

itself. The data structures and the implementation details of the algorithm, as well as the final phase of the preprocessing for shortest path queries, are presented in Section 5. We close in Section 7 with a discussion, which includes the extension to the construction of *geodesic Voronoi diagrams* on ∂P, and with several open problems.

2 A Conforming Surface Subdivision

The input to our shortest path problem is a convex polytope P with n vertices and a source point $s \in \partial P$. Without loss of generality, we assume that s is a vertex of P and that all facets of P are triangles, since more complex polytope facets can be triangulated in overall $O(n)$ time, and the number of edges introduced is linear in the number of vertices. We also assume that no edge of P is axis-parallel, since otherwise the polytope can be rotated in $O(n)$ time to enforce this property. Our model of computation is the real RAM.

A key ingredient of the algorithm is a special subdivision S of ∂P into cells, so that each cell $c \in S$ is bounded by $O(1)$ subdivision edges, and the *unfolding* of each subdivision edge $e \in \partial c$, at the polytope edges that it traverses, is a straight segment; see Section 2.1 for precise definitions.

We construct S in two steps. The first step builds a rectilinear oct-tree-like subdivision S_{3D} of $\mathbb{R}^3$ by taking into account only the vertices of P (and the source point s); the second step intersects ∂P with the subfaces of S_{3D}. These intersections define (though do not coincide with) the surface subdivision edges, thereby yielding an (implicit) representation of S.

The algorithm for the first step (constructing a "conforming" 3-dimensional subdivision for a set of points) is somewhat complicated on one hand, and very similar to the corresponding construction in [22] on the other hand (except for the modifications needed to handle the spatial situation). It is also quite independent of the main part of the shortest path algorithm, and so we postpone its presentation to Section 6 at the end of the paper. In the present section, we only state the properties that S_{3D} should satisfy, assume that it is already available, and describe how to use it for constructing S. We start with some preliminary definitions.

2.1 Preliminaries

We borrow some definitions from [30, 39, 40]. A *geodesic path* π is a simple (that is, not self-intersecting) path along ∂P that is locally optimal, in the sense that, for any two sufficiently close points $p, q \in \pi$, the portion of π between p and q is the unique shortest path that connects them on ∂P. Such a path π is always piecewise linear; its length is defined as the sum of the lengths of all its straight segments, and is denoted as $|\pi|$. For any two points $a, b \in \partial P$, a *shortest geodesic path* between them is denoted by $\pi(a, b)$. Generally, $\pi(a, b)$ is unique, but there are degenerate placements of a and b for which there exist several geodesic shortest paths that connect them. For convenience, the word "geodesic" is omitted in the

rest of the paper. For any two points $a, b \in \partial P$, at least one shortest path $\pi(a, b)$ exists [30]. We use the notation $\Pi(a, b)$ to denote the *set of shortest paths* connecting a and b. The length of any path in $\Pi(a, b)$ is the shortest path distance between a and b, and is denoted as $d_S(a, b)$. We occasionally use $d_S(X, Y)$ to denote the shortest path distance between two compact sets of points $X, Y \subseteq \partial P$, which is the minimum $d_S(x, y)$, over all $x \in X$ and $y \in Y$. We use $d_{3D}(x, y)$ to denote the Euclidean distance in $\mathbb{R}^3$ between two points x and y, and $d_{3D}(X, Y)$ is also occasionally used to denote the (analogously defined) Euclidean distance in $\mathbb{R}^3$ between two sets of points X, Y. When considering points x, y on a plane, we sometimes denote $d_{3D}(x, y)$ by $d(x, y)$. The notation $d_\infty(x, y)$ denotes the distance between x and y under the L_∞ norm.

If facets f and f' share a common edge χ, the *unfolding* of f' onto (the plane containing) f is the rigid transformation that maps f' into the plane containing f, effected by an appropriate rotation about the line through χ, so that f and the image of f' lie on opposite sides of that line. Let $\mathcal{F} = (f_0, f_1, \ldots, f_k)$ be a sequence of distinct facets such that f_{i-1} and f_i have a common edge χ_i, for $i = 1, \ldots, k$. We say that $\mathcal{F}$ is the *corresponding facet sequence* of the *edge sequence* $\mathcal{E} = (\chi_1, \chi_2, \ldots, \chi_k)$, and that $\mathcal{E}$ is the *corresponding edge sequence* of $\mathcal{F}$. The unfolding transformation $U_\mathcal{E}$ is the transformation of 3-space that represents the rigid motion that maps f_0 to the plane of f_k, through a sequence of unfoldings at the edges $\chi_1, \chi_2, \ldots, \chi_k$. That is, for $i = 1, \ldots, k$, let φ_i be the rigid transformation of 3-space that unfolds f_{i-1} to the plane of f_i about χ_i. The unfolding $U_\mathcal{E}$ is then the composed transformation $\Phi_\mathcal{E} = \varphi_k \circ \varphi_{k-1} \circ \ldots \circ \varphi_1$. The unfolding of an empty edge sequence is the identity transformation.

However, in what follows, we will also use $U_\mathcal{E}$ to denote the collection of *all partial unfoldings* $\Phi_\mathcal{E}^{(i)} = \varphi_k \circ \varphi_{k-1} \circ \ldots \circ \varphi_i$, for $i = 1, \ldots, k$. Thus $\Phi_\mathcal{E}^{(i)}$ is the unfolding of f_{i-1} onto the plane of f_k. The *domain* of $U_\mathcal{E}$ is then defined as the union of all points in $f_0, f_1, \ldots, f_k$, and the plane of the last facet f_k is denoted as the *destination plane* of $U_\mathcal{E}$. See Figure 2.

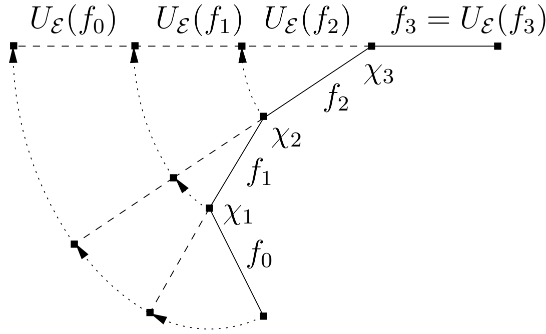

Figure 2: *Side view: the unfolding of the facet sequence $\mathcal{F} = (f_0, f_1, f_2, f_3)$. The corresponding edge sequence is $\mathcal{E} = (\chi_1, \chi_2, \chi_3)$, and we denote by $U_\mathcal{E}$ the following collection of transformations: (i) φ_3 that unfolds f_2 to the plane of f_3 about χ_3, (ii) $\varphi_3 \circ \varphi_2$, where φ_2 unfolds f_1 to the plane of f_2 about χ_2, and (iii) $\varphi_3 \circ \varphi_2 \circ \varphi_1$, where φ_1 unfolds f_0 to the plane of f_1 about χ_1.*

Each rigid transformation in $\mathbb{R}^3$ can be represented as a 4×4 matrix,[3] so the entire sequence $\Phi_\mathcal{E} = \Phi_\mathcal{E}^{(1)}, \Phi_\mathcal{E}^{(2)}, \ldots, \Phi_\mathcal{E}^{(k)}$ can be computed in $O(k)$ time.

[3]Specifically, assume that the current coordinate frame C is centered at a vertex u of P, so that the x-axis

The unfolding $U_{\mathcal{E}}(\mathcal{F})$ of the facet sequence $\mathcal{F}$ is the union $\bigcup_{i=0}^{k} \Phi_{\mathcal{E}}^{(i+1)}(f_i)$ of the unfoldings of each of the facets $f_i \in \mathcal{F}$, in the destination plane of $U_{\mathcal{E}}$ (here the unfolding transformation for f_k is the identity). The unfolding $U_{\mathcal{E}}(\pi)$ of a path $\pi \subset \partial P$ that traverses the edge sequence $\mathcal{E}$, is the path consisting of the unfolded images of all the points of π in the destination plane of $U_{\mathcal{E}}$.

Note that our definition of unfolding is asymmetric, in the sense that we could equally unfold into the plane of any of the other facets of $\mathcal{F}$. We sometimes ignore the exact choice of the destination plane, since the appropriate rigid transformation that moves between these planes is easy to compute.

Remark: In the general case, the unfolded image of a facet sequence (or of a path) might *overlap* itself. To maintain correctly the geometry of the unfolded image, we treat the unfolded surface as a *Riemann surface*, where the overlapping does not occur, because points that overlap lie in different "layers" of the surface. See Section 3 for a detailed discussion of this issue.

The following properties of shortest paths are proved in [10, 30, 39, 40].

(i) A shortest path π on ∂P does not traverse any facet of P more than once; that is, the intersection of π with any facet f of ∂P is a (possibly empty) line segment.

(ii) If π traverses the edge sequence $\mathcal{E}$, then the unfolded image $U_{\mathcal{E}}(\pi)$ is a straight line segment.

(iii) A shortest path π never crosses a vertex of P (but it may start or end at a vertex).

(iv) For any three distinct points $a, b, c \in \partial P$, either one of the shortest paths $\pi(a, b), \pi(a, c)$ is a subpath of the other, or these two paths meet only at a. For any four distinct points $a, b, c, d \in \partial P$, the shortest paths $\pi(a, b), \pi(c, d)$ intersect in at most one point, unless one of these paths is a subpath of the other, or their intersection is a shortest path between one of the points a, b and one of the points c, d. In other words, two shortest

contains the polytope edge $\overline{uw}$ and the facet $f = \triangle uvw$ is contained in the xy-plane. Denote by $C(p)$ the 4-vector of the homogeneous coordinates of a point p in the frame C. Let C' be a new coordinate frame centered at v so that the new x-axis contains the polytope edge $\overline{vw}$ and the new xy-plane contains the other facet f' bounded by $\overline{vw}$. Then $C'(p) = FC(p)$, where $F = R(x, \theta)R(z, \alpha)T(v)$, θ is the angle of rotation about the line through $\overline{vw}$, so that f' and the image of f become coplanar and lie on opposite sides of that line, α is the angle $\measuredangle uwv$, and $R(x, \theta), R(z, \alpha), T(v)$ are the following matrices: $R(x, \theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ rotates by θ around the (current) x-axis, $R(z, \alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ rotates by α around the (current) z-axis, and $T(v) = \begin{pmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ translates $C(p)$ by the vector $\overline{vu} = \begin{pmatrix} v_x \\ v_y \\ 0 \\ 0 \end{pmatrix}$. See [37] for details.

paths from the same source point s, so that none of them is an extension of the other, cannot intersect each other except at s and, if they have the same destination point, possibly at that point too.

2.1.1 The elements of the shortest path map

In this subsection we discuss the structure of the *shortest path map*, which our algorithm aims to compute (implicitly).

We consider the problem of computing shortest paths from a fixed *source* point $s \in \partial P$ to all points of ∂P. A point $z \in \partial P$ is called a *ridge* point if there exist at least two distinct shortest paths from s to z. The *shortest path map* with respect to s, denoted $\mathrm{SPM}(s)$, is a subdivision of ∂P into at most n connected regions, called *peels*, whose interiors are vertex-free, and contain neither ridge points nor points belonging to shortest paths from s to vertices of P, and such that for each such region Φ, there is only one shortest path $\pi(s,p) \in \Pi(s,p)$ to any $p \in \Phi$, which also satisfies $\pi(s,p) \subset \Phi$.

The following properties of ridge points are proved in [40].

(i) A shortest path from s to any point in ∂P cannot pass through a ridge point (but it may end at such a point).

(ii) The set of all ridge points is the union of $O(n^2)$ straight segments.

(iii) The set of all vertices of P and ridge points is a tree having (some of) the vertices of P as leaves (there are degenerate cases where a vertex of P is an internal node in the tree).

See Figure 3 for an illustration.

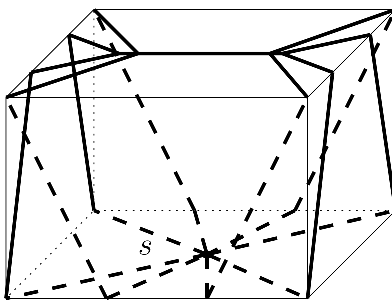

Figure 3: *Peels are bounded by thick lines (dashed and solid). The bisectors (the set of all the ridge points) are the thick solid lines, while the dashed solid lines are the virtual edges of* $\mathrm{SPM}(s)$ *that lead to the vertices of* P.

Remark: Property (ii) suggests that in the worst case, the space complexity of an *explicit* representation of $\mathrm{SPM}(s)$ might be quadratic. Indeed, this may be the case when every peel intersects $\Omega(n)$ facets of ∂P, a situation that is easy to realize. Hence, to keep the complexity of our algorithm close to linear, $\mathrm{SPM}(s)$ is never computed explicitly. Instead, the algorithm

collects data that represents SPM(s) *implicitly*, while still allowing us to answer shortest path queries (with respect to s) efficiently — see Sections 4 and 5 for details.

There are two types of vertices of SPM(s):

(i) A ridge point that is incident to three or more peels.

(ii) A vertex of P, including s.

The boundaries of the peels form the *edges* of SPM(s). There are two types of edges (see Figure 3):

(i) A maximal connected polygonal path of ridge points between two vertices of SPM(s), that does not contain any vertex of SPM(s), is called a *bisector*.

(ii) A shortest path from s to a vertex of P, is called a *virtual edge* of SPM(s). (Assuming general position, each vertex of P has a unique shortest path from s.)

It is proved in [40] that SPM(s) has only $O(n)$ vertices and (folded) edges (each of these edges potentially breaks down, when folded, into $O(n)$ straight segments).

Remark: An explicit representation of the *folded* SPM(s) can have $\Theta(n^2)$ complexity, which we definitely do not want to produce. In contrast, an explicit representation of the *unfolded* map (let us call it a semi-explicit representation) has only $O(n)$ complexity, and could be useful for various purposes. Our algorithm does not produce such a semi-explicit representation (its implicit representation is looser — see Section 5.4), but we believe that it can be modified to construct the "unfolded SPM(s)" as a byproduct, by adapting the idea of Hershberger and Suri [22] of explicitly detecting all wave collisions using "artificial wavefronts". We leave it as an interesting open question whether such a semi-explicit representation can be further processed in near-linear time to enable efficient shortest-path queries.[4]

Denote by $\mathcal{E}_i$ the *maximal polytope edge sequence* crossed by a shortest path from s to a vertex of a peel Φ_i inside Φ_i. (Notice that a maximal polytope edge sequence of a given peel is unique, since a peel does not contain polytope vertices in its interior.) Denote by s_i the unfolded source image $U_{\mathcal{E}_i}(s)$; for the sake of simplicity, we use the same notation s_i to also denote the unfolded source image $U_{\mathcal{E}'_i}(s)$, for any prefix $\mathcal{E}'_i$ of $\mathcal{E}_i$. A bisector between two adjacent peels Φ_i, Φ_j is denoted by $b(s_i, s_j)$. It is the locus of points q equidistant from s_i and s_j (on some common plane), so that there are at least two shortest paths in $\Pi(s, q)$ — one, completely contained in Φ_i, traverses a prefix of the polytope edge sequence $\mathcal{E}_i$, and the other, completely contained in Φ_j, traverses a prefix of the polytope edge sequence $\mathcal{E}_j$.

Note that for two maximal polytope edge sequences $\mathcal{E}_i, \mathcal{E}_j$, the bisector $b(s_i, s_j)$ between the source images $s_i = U_{\mathcal{E}_i}(s)$ and $s_j = U_{\mathcal{E}_j}(s)$ satisfies both the following properties: $U_{\mathcal{E}_i}(b(s_i, s_j)) \subset U_{\mathcal{E}_i}(\mathcal{F}_i)$, and $U_{\mathcal{E}_j}(b(s_i, s_j)) \subset U_{\mathcal{E}_j}(\mathcal{F}_j)$, where $\mathcal{F}_i, \mathcal{F}_j$ are the corresponding facet sequences of $\mathcal{E}_i, \mathcal{E}_j$, respectively.

[4] In [11] this is done in at least quadratic time.

2.2 The 3-dimensional subdivision and its properties

We begin by introducing the subdivision S_{3D} of $\mathbb{R}^3$, whose construction is given in Section 6. The subdivision is composed of 3D-cells, each of which is either a whole axis-parallel cube or an axis-parallel cube with a single axis-parallel cube-shaped hole (we then call it a *perforated* cube). The boundary face of each 3D-cell is divided into either 16×16 or 64×64 square subfaces with axis-parallel sides.[5] See Figure 4 for an illustration.

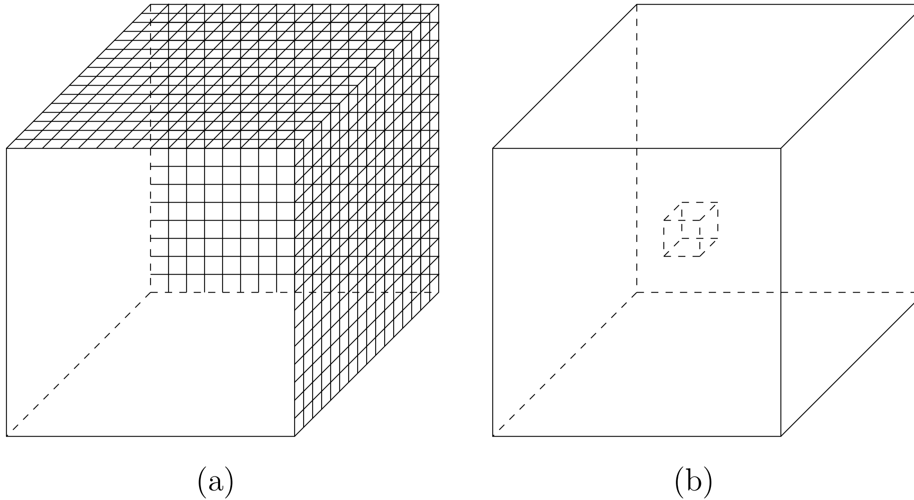

Figure 4: *Two types of a 3D-cell: (a) A whole cube, where the subdivision of three of its faces is shown. (b) A perforated cube. Each of its faces (both inner and outer) is subdivided into subfaces (not shown).*

Let $l(h)$ denote the edge length of a square subface h.

The crucial property of S_{3D} is the *well-covering* of its subfaces. Specifically, a subface h of S_{3D} is said to be *well-covered* if the following three conditions hold:

(W1) There exists a set of $O(1)$ cells $C(h) \subseteq S_{3D}$ such that h lies in the interior of their union $R(h) = \bigcup_{c \in C(h)} c$. The region $R(h)$ is called the *well-covering region* of h (see Figure 5 for an illustration).

(W2) The total complexity of the subdivisions of the boundaries of all the cells in $C(h)$ is $O(1)$.

(W3) If g is a subface on $\partial R(h)$, then $d_{3D}(h, g) \geq 16 \max\{l(h), l(g)\}$.

A subface h is *strongly well-covered* if the stronger condition (W3') holds:

(W3') For any subface g so that h and g are portions of nonadjacent (undivided) faces of the subdivision, $d_{3D}(h, g) \geq 16 \max\{l(h), l(g)\}$.

[5] With some care, one can improve the constants to 12×12 and 48×48, respectively, by a trivial modification of the construction of S_{3D}. However, this will require more work on the 3D-cells that contain the vertices of P, to keep the minimum vertex clearance property, defined below; we therefore skip this optimization for the sake of simplicity.

Remark: The wavefront propagation algorithm described in Sections 4, 5 requires the subfaces of S_{3D} to be only well-covered, but not necessarily strongly well-covered. The stronger condition (W3') of subfaces of S_{3D} is only needed in the construction of the surface subdivision S, described in the next subsection.

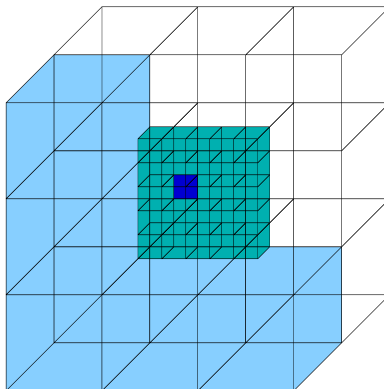

Figure 5: *The well-covering region of the darkly shaded face h contains, in this example, a total of 39 3D-cells (nine transparent large cells on the back, five lightly shaded large cells on the front, and 25 small cells, also on the front). Each face of the boundary of each 3D-cell in this figure is further subdivided into smaller subfaces (not shown). The well-covering region of each of the subfaces of h coincides with $R(h)$.*

Let V denote the set of vertices of the polytope (including the source vertex s). A 3D-subdivision S_{3D} is called a *(strongly) conforming 3D-subdivision* for V if the following three conditions hold.

(C1) Each cell of S_{3D} contains at most one point of V in its closure.

(C2) Each subface of S_{3D} is (strongly) well-covered.

(C3) The well-covering region of every subface of S_{3D} contains at most one vertex of V.

Remarks:

(i) The subdivision is called *conforming* because conditions (C1) and (C3) force it to "conform" to the distribution of points in V, encapsulating each point in a separate cell, which are "reasonably far" from each other.

(ii) The 3D-subdivision S_{3D} is similar to a (compressed) oct-tree in that all its faces are axis-parallel and their sizes grow by factors of 4. However, the cells of S_{3D} may be nonconvex and the union of the surfaces of the 3D-subdivision itself may be disconnected.

(iii) We actually require property (C2) to hold only for each *internal* subface of S_{3D}, that is, only for subfaces that bound two 3D-cells (rather than one). This is because the *external* subfaces do not intersect P, and therefore are not involved in the construction of the surface subdivision S. However, in the rest of the paper we ignore this minor detail, for the sake of simplicity.

S_{3D} also has the following *minimum vertex clearance property*:

(MVC) For any point $v \in V$ and for any subface h, $d_{3D}(v, h) \geq 4l(h)$.

As mentioned, the algorithm for computing a strongly conforming 3D-subdivision of V is presented in Section 6. We state the main result shown there.[6]

Theorem 2.1 (Conforming 3D-subdivision Theorem). *Every set of n points in $\mathbb{R}^3$ admits a strongly conforming 3D-subdivision S_{3D} of $O(n)$ size, that also satisfies the minimum vertex clearance property. In addition, each input point is contained in the interior of a distinct whole cube cell. Such a 3D-subdivision can be constructed in $O(n \log n)$ time.*

2.3 Computing the surface subdivision

We form the surface subdivision S from the 3D-subdivision S_{3D}, as follows. We intersect the edges of S_{3D} with ∂P: Points where those edges cross ∂P are called *the transparent endpoints* of S. Then, we use the transparent endpoints to define the edges of S. Informally (a formal treatment follows shortly), we replace each intersection ξ of a subface of S_{3D} with ∂P by a shortest among all paths on ∂P that connect the endpoints of ξ and traverse the same facet sequence as ξ. Thus, when appropriately unfolded, transparent edges become straight segments. As a result, we get two types of edges on ∂P: the edges of the surface-subdivision S, that we call *transparent edges*, in accordance with the notation of [22], and the original *polytope edges*. (We usually use the letters e, f, g to denote transparent edges, and the letter χ to denote polytope edges.) The algorithm uses the transparent edges as "stepping stones" for the Dijkstra-style wavefront propagation process, where each step in this process propagates wavefronts from a transparent edge to $O(1)$ other transparent edges of nearby cells. A major technical issue that our algorithm has to face, which is absent in the planar algorithm of [22], is that a transparent edge may traverse many (up to $\Theta(n)$) facets of P, but we still want to treat it as a single entity. This will force the algorithm to use a compact representation of transparent edges, which will be described later. In contrast, in the planar case the transparent edges are simply straight segments, which are trivial to represent and to manipulate.

The transparent edges of S induce a partition of ∂P into 2-dimensional connected regions, called the *surface cells* of S. Each surface cell of S *originates* from a 3D-cell of S_{3D}. We use the term "originate", because the boundary of a surface cell c is close to (in the sense explained later in this section), but not identical to, the corresponding intersection of ∂P with the 3D-cell that c originates from. A surface cell originates from exactly one 3D-cell of S_{3D}, but a 3D-cell may have more than one surface cell originating from it. See Figure 6 for a schematic illustration of this phenomenon.

We first describe the construction of S using the conforming 3D-subdivision S_{3D}. Then we analyze and establish several properties of S. The running time of the construction, which is $O(n \log n)$, is established later in Lemma 2.11.

Transparent edges. We intersect the subfaces of S_{3D} with ∂P. Each maximal connected portion ξ of the intersection of a subface h of S_{3D} with ∂P induces *a surface-subdivision*

[6]Note that we do not assume that the points of V are in convex position.

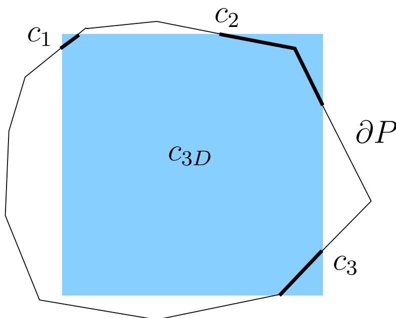

Figure 6: *A two-dimensional illustration of three surface cells c_1, c_2, c_3 originating from a single 3D-cell c_{3D}.*

(transparent) edge e of S with the same pair of endpoints. (We emphasize again that e is in general different from ξ. The precise construction of e is detailed below.) A single subface h can therefore induce up to four transparent edges (since P is convex and h is a square, and the construction of S_{3D} ensures that none of its edges is incident to a polytope edge; see Figure 7). Isolated points of such an intersection are ignored in the construction (in fact, assuming general position, no isolated points will arise). If ξ is a closed cycle fully contained in the interior of h, we break it at its x-rightmost and x-leftmost points (or y-rightmost and y-leftmost points, if h is perpendicular to the x-axis). These two points are regarded as two new endpoints of transparent edges. These endpoints, as well as the endpoints of the open connected intersection portions ξ, are referred to as *transparent endpoints*.

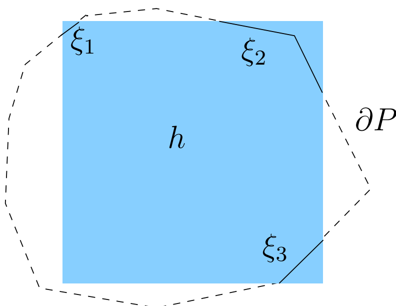

Figure 7: *A subface h and three maximal connected portions ξ_1, ξ_2, ξ_3 that constitute the intersection $h \cap \partial P$.*

Let $\xi(a, b)$ be a maximal connected portion of the intersection of a subface h of S_{3D} with ∂P, bounded by two transparent endpoints a, b. Let $\mathcal{E} = \mathcal{E}_{a,b}$ denote the sequence of polytope edges that $\xi(a, b)$ crosses from a to b, and let $\mathcal{F} = \mathcal{F}_{a,b}$ denote the facet sequence corresponding to $\mathcal{E}$. We define the *transparent edge* $e_{a,b}$ as the shortest path from a to b *within* the union of $\mathcal{F}$ (a priori, $U_{\mathcal{E}}(e_{a,b})$ needs not be a straight segment, but we will shortly show that it is). We say that $e_{a,b}$ *originates from the cut* $\xi(a, b)$. Obviously, its length $|e_{a,b}|$ is equal to $|U_{\mathcal{E}}(e_{a,b})| \leq |\xi(a, b)|$. See Figure 8 for an illustration. Note that in general position (in particular, if no polytope edge is axis-parallel), $e_{a,b} = \xi(a, b)$ if and only if $\xi(a, b)$ is contained in exactly one facet of P. (This initial collection of transparent edges may contain crossing pairs, and each initial transparent edge will be split into sub-edges at the points

where other edges cross it — see below.)

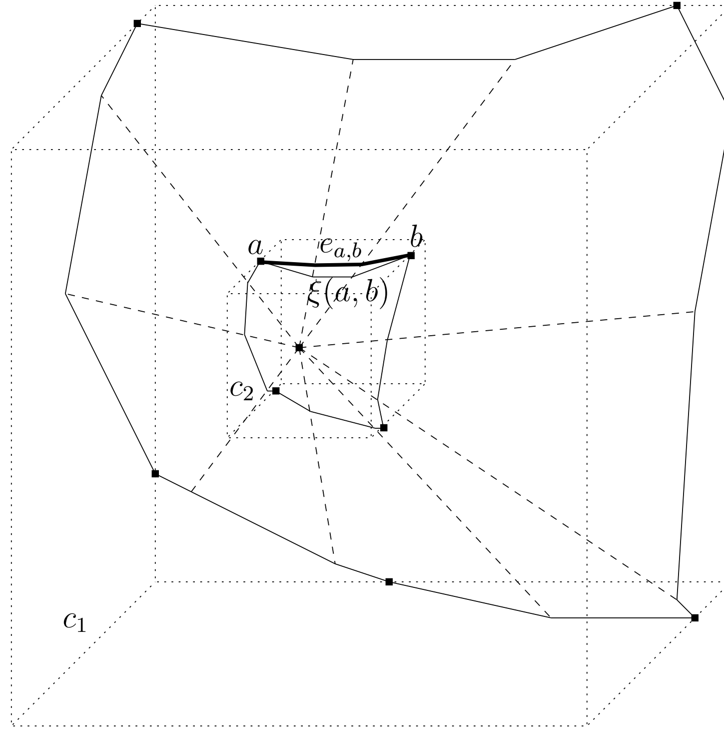

Figure 8: *The 3D-cells c_1 and c_2 are denoted by dotted lines. The cuts of their boundaries with ∂P are denoted by thin solid lines, and the dashed lines denote polytope edges. (To simplify the illustration, it is ignored in this figure that the faces of S_{3D} are actually subdivided into smaller subfaces.)*

Lemma 2.2. *No polytope vertex can be incident to transparent edges. That is, for each transparent edge $e_{a,b}$, the unfolded path $U_{\mathcal{E}}(e_{a,b})$ is a* straight segment*.*

Proof: By the minimum vertex clearance property, for any subface h of S_{3D} and for any $v \in V$, we have $d_{3D}(h, v) \geq 4l(h)$. Let $e_{a,b}$ be a transparent edge originating from $\xi(a, b) \subset h \cap \partial P$. Then $|e_{a,b}| \leq |\xi(a, b)|$, by definition of transparent edges, and $|\xi(a, b)| \leq 4l(h)$, since $\xi(a, b) \subseteq h$ is convex, and h is a square of side length $l(h)$. Therefore $d_{3D}(a, v) \geq |e_{a,b}|$, which shows that $e_{a,b}$ cannot reach any vertex v of P. $\square$

Lemma 2.3. *A transparent endpoint is incident to at least two and at most $O(1)$ transparent edges.*

Proof: A transparent edge endpoint x either delimits two portions of a cyclic cut, or, in general position, is incident to exactly one edge e_{3D} of S_{3D}. In the former case x is incident to exactly two transparent edges; in the latter case e_{3D} bounds between two and four subfaces that intersect ∂P at x, and the claim follows. (It is easy to see that even without the general position assumption x is still incident to only $O(1)$ edges of S_{3D}.) $\square$

Lemma 2.4. *Each transparent edge that originates from some face ϕ of S_{3D}, meets at most $O(1)$ other transparent edges that originate from faces of S_{3D} adjacent to ϕ (or from ϕ itself),*

and does not cross any other transparent edges (that originate from faces of S_{3D} not adjacent to ϕ).

Proof: Let $e_{a,b}$ be a transparent edge originating from the cut $\xi(a,b)$, and let $e_{c,d}$ be a transparent edge originating from the cut $\xi(c,d)$. Let h, g be the subfaces of S_{3D} that contain $\xi(a,b)$ and $\xi(c,d)$, respectively. Since $a, b \in h$, the 3D-distance from any point of $e_{a,b}$ to h is at most $\frac{1}{2}|e_{a,b}| \le \frac{1}{2}|\xi(a,b)| \le 2l(h)$. Similarly, the 3D-distance from any point of $e_{c,d}$ to g is no larger than $2l(g)$. Recall that S_{3D} is a strongly conforming 3D-subdivision. Therefore, if h, g are incident to non-adjacent faces of S_{3D}, then, by (W3'), $d_{3D}(h,g) \ge 16\max\{l(h), l(g)\}$, hence $e_{a,b}$ does not intersect $e_{c,d}$. There are only $O(1)$ faces of S_{3D} that are adjacent to the facet of h, and each of them contains $O(1)$ subfaces g. Hence there are at most $O(1)$ possible choices of g for each h, and the first part of the claim follows. $\square$

Splitting intersecting transparent edges. Crossing transparent edges are illustrated in Figure 9. We first show how to compute the intersection points; then, each intersection point is regarded as a new transparent endpoint, splitting each of the two intersecting edges into sub-edges.

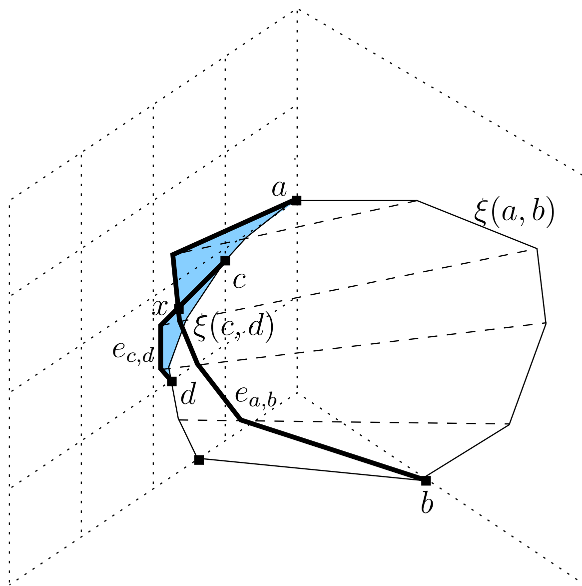

Figure 9: *Subfaces are bounded by dotted lines, polytope edges are dashed, the cuts of $\partial P \cap S_{3D}$ are thin solid lines, and the two transparent edges $e_{a,b}, e_{c,d}$ are drawn as thick solid lines. The edges $e_{a,b}, e_{c,d}$ intersect each other at the point $x \in \partial P$; the shaded region of ∂P (including the point x on its boundary) lies in this illustration beyond the plane that contains the cut $\xi(c,d)$.*

Lemma 2.5. *A maximal contiguous facet subsequence that is traversed by a pair of intersecting transparent edges e, e' contains either none or only one intersection point of $e \cap e'$. In the latter case, it contains an endpoint of e or e' (see Figure 10).*

Proof: Consider some maximal common facet subsequence $\tilde{\mathcal{F}} = (f_0, \ldots, f_k)$ that is traversed by e and e', so that the union R of the facets in $\tilde{\mathcal{F}}$ contains an intersection point of $e \cap e'$. Since $\tilde{\mathcal{F}}$ is maximal, no edge of ∂R is crossed by both e and e'; in particular, $\tilde{\mathcal{F}}$ cannot be a

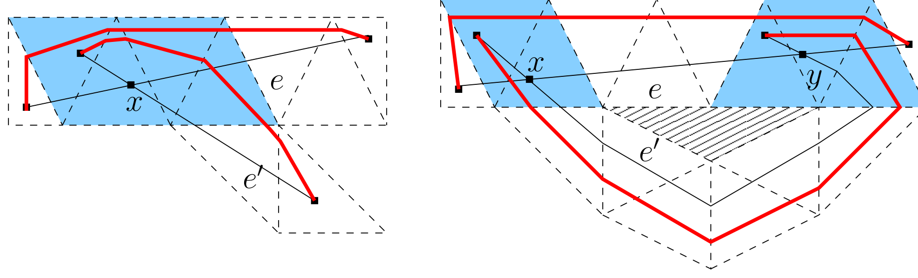

Figure 10: *A top view of two examples of intersecting transparent edges e, e' (thin solid lines); the corresponding original cuts (thick solid lines) never intersect each other. The maximal contiguous facet subsequences that are traversed by both e, e' and contain an intersection point of $e \cap e'$ are shaded. In the second example, the "hole" of ∂P between the facet sequence traversed by e and the facet sequence traversed by e' is hatched.*

single triangle, so $k \geq 1$. Since e and e' are shortest paths within R, they cannot cross each other (within R) more than once, which proves the first part of the lemma.

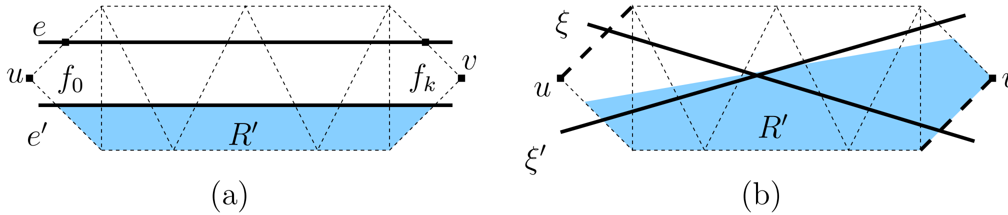

Figure 11: *R is the union of all the facets. (a) e' divides R into two regions, one of which, R' (shaded), contains neither u nor v. (b) If R' contains v but not u, ξ' (crossing the same edge sequence as e') intersects ξ (which must cross the bold dashed edges, since R is maximal).*

To prove the second claim, assume the contrary — that is, R does not contain any endpoint of e and of e'. Denote by u (resp., v) the vertex of f_0 (resp., f_k) that is not incident to f_1 (resp., f_{k-1}). We claim that e' divides R into two regions, one of which contains both u and v, and the other, which we denote by R', contains neither u nor v. Indeed, if each of the two subregions contained exactly one point from $\{u, v\}$ then, by maximality of $\tilde{\mathcal{F}}$, e and e' would have to traverse facet sequences that "cross" each other, which would have forced the corresponding original cuts ξ, ξ' also to cross each other, contrary to the construction; see Figure 11. The transparent edge e intersects ∂R in exactly two points that are not incident to R'. Since e intersects e' in R, e must intersect $\partial R' \cap e'$ in two points — a contradiction. □

By Lemma 2.4, each transparent edge e has at most $O(1)$ candidate edges that can intersect it (at most four times, as follows from Lemma 2.5). For each such candidate edge e', we can find each of the four possible intersection points, using Lemma 2.5, as follows. First, we check for each of the extreme facets in the facet sequence traversed by e, whether it is also traversed by e', and vice versa (if all the four tests are negative, then e and e' do not intersect each other). We describe in the proof of Lemma 2.11 below how to perform these tests efficiently. For each positive test — when a facet f that is extreme in the facet sequence traversed by one of e, e', is present in the facet sequence traversed by the other — we unfold

both e, e' to the plane of f, and find the (image in the plane of f of the) intersection point of $e \cap e'$ that is closest to f (among the two possible intersection points).

Surface cells. After splitting the intersecting transparent edges, the resulting transparent edges are pairwise openly disjoint and subdivide ∂P into connected (albeit not necessarily simply connected) regions bounded by cycles of transparent edges, as follows from Lemma 2.3. These regions, which we call *surface cells*, form a planar (or, rather, spherical) map S on ∂P, which is referred to as the *surface subdivision* of P that is induced by S_{3D}. Each surface cell is bounded by a set of cycles of transparent edges that are induced by some 3D-cell c_{3D}, and possibly also by a set of other 3D-cells adjacent to c_{3D} whose originally induced transparent edges split the edges originally induced by c_{3D}.

Corollary 2.6. *Each 3D-cell induces at most $O(1)$ (split) transparent edges.*

Proof: Follows immediately from the property that the boundary of each 3D-cell consists of only $O(1)$ subfaces, from the fact that each subface induces up to four transparent edges, and from Lemmas 2.4 and 2.5. □

Corollary 2.7. *For each surface cell, all transparent edges on its boundary are induced by $O(1)$ 3D-cells.*

Proof: Follows immediately from Lemma 2.4. □

Corollary 2.8. *Each surface cell is bounded by $O(1)$ transparent edges.*

Proof: Follows immediately from Corollaries 2.6 and 2.7. □

In particular, Corollary 2.8 also shows that the boundary of each surface cell consists of only $O(1)$ cycles (connected components) of transparent edges. In fact, it is easy to check that there can be at most six (resp., eight) such cycles of edges that are induced by the outer (resp., inner) boundary of a 3D-cell; although we do not prove it formally, the explanation is illustrated in Figure 12.

Well-covering. We require that *all transparent edges be well-covered* in the surface subdivision S (compare to the well-covering property of the subfaces of S_{3D}), in the following modified sense.

- ($W1_S$) For each transparent edge e of S, there exists a set $C(e)$ of $O(1)$ cells of S such that e lies in the interior of their union $R(e) = \bigcup_{c \in C(e)} c$, which is referred to as the *well-covering region* of e.
- ($W2_S$) The total complexity of all the cells in $C(e)$ is $O(1)$.
- ($W3_S$) Let e_1 and e_2 be two transparent edges of S such that e_2 lies on the boundary of the well-covering region $R(e_1)$. Then $d_S(e_1, e_2) \geq 2 \max\{|e_1|, |e_2|\}$.

As the next theorem shows, our surface subdivision S is a *conforming surface subdivision* for P, in the sense that the following three properties hold.

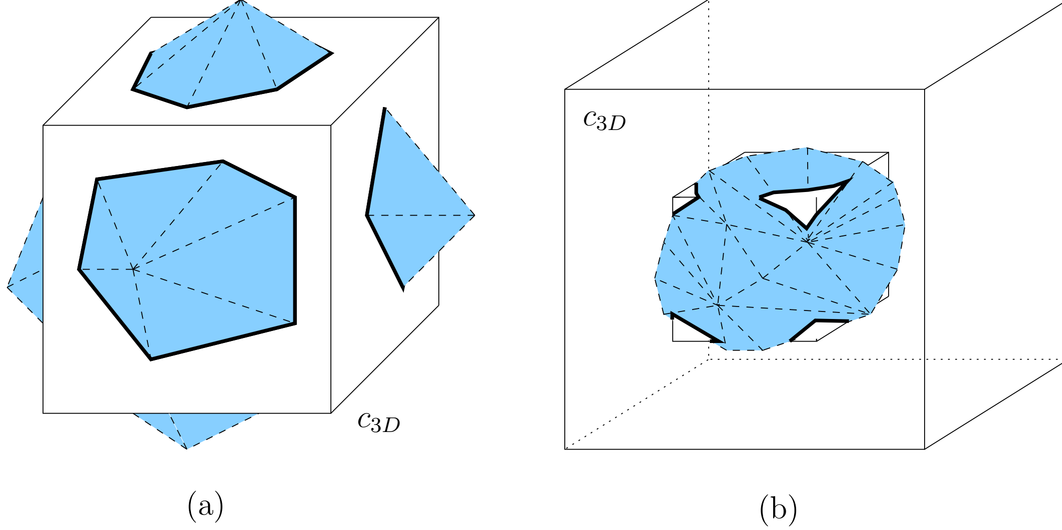

Figure 12: *P (shaded) intersects: (a) the outer boundary of a 3D-cell c_{3D} (white), forming at most six cycles of edges; (b) the inner boundary of a perforated cube c_{3D}, forming at most eight cycles of edges.*

($C1_S$) Each cell of S is a region on ∂P that contains at most one vertex of P in its closure.

($C2_S$) Each edge of S is well-covered.

($C3_S$) The well-covering region of every edge of S contains at most one vertex of P.

Theorem 2.9 (Conforming Surface-Subdivision Theorem). *Each convex polytope P with n vertices in $\mathbb{R}^3$ admits a conforming surface subdivision S of $O(n)$ size, constructed as described above.*

Proof: The properties ($C1_S$), ($C3_S$) follow from the properties (C1), (C3) of S_{3D}, respectively, and from the fact that each cycle $\mathcal{C}$ of transparent edges that forms a connected component of the boundary of some cell of S traverses the same polytope edge sequence as the original intersections of S_{3D} with ∂P that induce $\mathcal{C}$.

To show well-covering of edges of S (property ($C2_S$)), consider an original transparent edge $e_{a,b}$ (before the splitting of intersecting edges). The endpoints a, b are incident to some subface h which is well-covered in S_{3D}, by a region $R(h)$ consisting of $O(1)$ 3D-cells. We define the well-covering region $R(e)$ of every edge e, obtained from $e_{a,b}$ by splitting, as the connected component containing e, of the union of the surface cells that originate from the 3D-cells of $R(h)$. There are clearly $O(1)$ surface cells in $R(e)$, since each 3D-cell of S_{3D} induces at most $O(1)$ (transparent edges that bound at most $O(1)$) surface cells. $R(e)$ is not empty and it contains e in its interior, since all the surface cells that are incident to e originate from 3D-cells that are incident to h and therefore are in $R(h)$. For each transparent edge e' originating from a subface g that lies on the boundary of (or outside) $R(h)$, $d_S(h, g) \geq d_{3D}(h, g) \geq 16 \max\{l(h), l(g)\}$. The length of e satisfies $|e| \leq |e_{a,b}| \leq |\xi(a, b)| \leq 4l(h)$, and, similarly, $|e'| \leq 4l(g)$. Therefore, for each $p \in e$ we have $d_{3D}(p, h) \leq 2l(h)$, and for each

$q \in e'$ we have $d_{3D}(q,g) \le 2l(g)$. Hence, for each $p \in e, q \in e'$, we have $d_S(p,q) \ge d_{3D}(p,q) \ge (16-4)\max\{l(h), l(g)\}$, and therefore $d_S(e,e') \ge 2\max\{|e|, |e'|\}$.[7] $\square$

We next simplify the subdivision by deleting each group of surface cells whose union completely covers exactly one hole of a single surface cell c and contains no vertices of P, thereby eliminating the hole and making it part of c. See Figure 13 for an illustration. This optimization clearly does not violate any of the properties of S proved above, and can be performed in $O(n)$ time. After the optimization, each hole of a surface cell of S must contain a vertex.

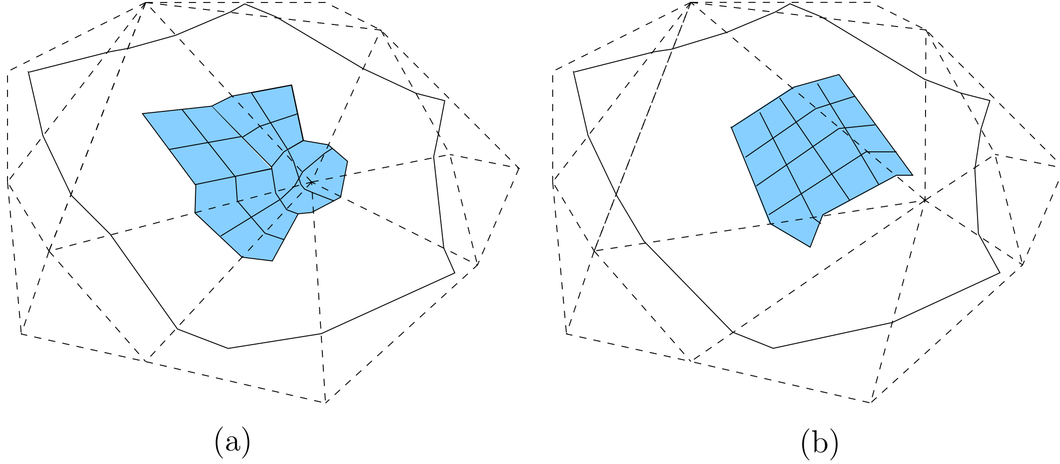

Figure 13: *Simplifying the subdivision (dashed edges denote polytope edges, and solid edges denote transparent edges). (a) None of the cells is discarded, since, although the shaded cells are completely contained inside a single hole of another cell, one of them contains a vertex of P. (b) All the shaded cells are discarded, and become part of the containing cell.*

The following lemma sharpens a simple property of S, which is used later in the analysis of surface unfoldings (see Section 3).

Lemma 2.10. *Let χ be a polytope edge, and let e be some transparent edge of S. Then e and χ intersect in at most one point.*

Proof: Follows immediately from the fact that e is a shortest path (within the union of a facet sequence). $\square$

2.4 The surface unfolding data structure

In this subsection we present the *surface unfolding data structure*, which we define and use for designing an efficient procedure for the construction of the surface subdivision. This data structure is also used in Section 3 to construct more complex data structures for wavefront propagation and in Section 5 for the wavefront propagation algorithm.

[7]In fact, we even have $d_S(e,e') \ge 3\max\{|e|, |e'|\}$; the gap between the required and the actual (tighter) bound follows from the fact that, as mentioned above, our 3D-subdivision contains more subfaces than necessary, which can be optimized with some care.

Sort the vertices of P in ascending z-order, and sweep a horizontal plane ζ upwards through P. At each height z of ζ, the cross section $P(z) = \zeta \cap P$ is a convex polygon, whose vertices are intersections of some polytope edges with ζ. The cross-section remains combinatorially unchanged, and each of its edges retains a fixed orientation, as long as ζ does not pass through a vertex of P. When ζ crosses a vertex v, the polytope edges incident to v and pointing downwards are deleted (as vertices) from $P(z)$, and those that leave v upwards are added to $P(z)$.

We can represent $P(z)$ by the circular sequence of its vertices, namely the circular sequence of the corresponding polytope edges. We use a linear, rather than a circular, sequence, starting with the x-rightmost vertex of $P(z)$ and proceeding counterclockwise along $\partial P(z)$. (It is easy to see that the rightmost vertex of $P(z)$ does not change as long as we do not sweep through a vertex of P.) We use a persistent search tree T_z (with path-copying, as in [23], for reasons detailed below) to represent the cross section. Since the total number of combinatorial changes in $P(z)$ is $O(n)$, the total storage required by T_z is $O(n \log n)$, and it can be constructed in $O(n \log n)$ time.

We construct, in a completely symmetric fashion, two additional persistent search trees T_x and T_y, by sweeping P with planes orthogonal to the x-axis and to the y-axis, respectively. They too use a total of $O(n \log n)$ storage and are constructed in $O(n \log n)$ time.

We can use the trees T_x, T_y, T_z to perform the following type of queries: Given an axis-parallel subface h of S_{3D} (or, more generally, any axis-parallel rectangle), compute efficiently the convex polygon $P \cap h$, and represent its boundary in compact form (without computing $P \cap h$ explicitly). Suppose, without loss of generality, that h is horizontal, say $h = [a, b] \times [c, d] \times \{z_1\}$. We access the value $T_z(z_1)$ of T_z at $z = z_1$ (which represents $P(z_1)$), and compute, in $O(\log n)$ time, the intersection points of the line $x = a$, $z = z_1$ with P (that is, with $P(z_1)$); there are at most two such intersection points. Similarly, we find the intersection points of each of the three other lines, supporting the edges of h, with P. It is easily seen that this can be done in a total of $O(\log n)$ time. We obtain at most eight intersection points, which partition $\partial P(z_1)$ into at most eight portions, and every other portion in the resulting sequence is contained in h. Since these are contiguous portions of $\partial P(z_1)$, each of them can be represented as the disjoint union of $O(\log n)$ subtrees of $T_z(z_1)$, where the endpoints of the portions (the intersection points of ∂h with $\partial P(z_1)$) do not appear in the subtrees, but can be computed explicitly in additional $O(1)$ time. Hence, we can compute, in $O(\log n)$ time, the polytope edge sequence of the intersection $P \cap h$, and represent it as the disjoint concatenation of $O(\log n)$ canonical sequences, each formed by the edges stored in some subtree of T_z. The trees T_x, T_y are used for handling, in a completely analogous manner, subfaces h orthogonal to the x-axis and to the y-axis, respectively.

We can also use T_z for another (simpler) type of query: Given a facet f of ∂P, locate the endpoints of $f \cap P(z)$ (which must be stored at two consecutive leaves in the cyclic order of leaves of T_z), or report that $f \cap P(z) = \emptyset$. As noted above, the *slopes* of the edges of $P(z)$ do not change when z varies, as long as $P(z)$ does not change combinatorially. Moreover, these slopes increase monotonically, as we traverse $P(z)$ in counterclockwise direction from its x-leftmost vertex v_L to its x-rightmost vertex v_R, and then again from v_R to v_L. This allows us to locate f in the sequence of edges of $P(z)$ by a binary search in the sequence of

their slopes, which takes $O(\log n)$ time. This search procedure is defined analogously in T_x and T_y.

However, the most important part of the structure is as follows. With each node ν of T_z, we precompute and store the unfolding U_ν of the sequence $\mathcal{E}_\nu$ of polytope edges stored at the leaves of the subtree of ν. This is done in an easy bottom-up fashion, exploiting the following obvious observation. Denote by $\mathcal{F}_\nu$ the corresponding facet sequence of $\mathcal{E}_\nu$. If ν_1, ν_2 are the left and the right children of ν, respectively, then the last facet in $\mathcal{F}_{\nu_1}$ coincides with the first facet of $\mathcal{F}_{\nu_2}$. Hence $U_\nu = U_{\nu_2} \circ U_{\nu_1}$, from which the bottom-up construction of all the unfoldings U_ν is straightforward. Each node stores exactly one rigid transformation, and each combinatorial change in $P(z)$ requires $O(\log n)$ transformation updates, along the path from the new leaf (or from the deleted leaf) to the root. (The rotations that keep the tree balanced do not inflate the time complexity; maintaining the unfolding information while rebalancing the tree is performed in a manner similar to that used in another related data structure, described in Section 5.1.) Hence the total number of transformations stored in T_z is $O(n \log n)$ (for all z, including the nodes added to the persistent tree with each path-copying), and they can all be constructed in $O(n \log n)$ time. Analogous constructions apply to T_x, T_y.

Let $\mathcal{F} = (f_0, f_1, \ldots, f_k)$ denote the corresponding facet sequence of the edge sequence that consists of the edges stored at the leaves of T_z at some fixed z. We next show how to use the tree T_z to perform another type of query: Compute the unfolded image $U(q)$ of some point $q \in f_i \in \mathcal{F}$ in the (destination) plane of some other facet $f_j \in \mathcal{F}$ (which is not necessarily the last facet of $\mathcal{F}$), and return the (implicit representation of) the corresponding edge sequence $\mathcal{E}_{ij}$ between f_i and f_j. If $i = j$, then $\mathcal{E}_{ij} = \emptyset$ and $U(q) = q$. Otherwise, we search for f_i and f_j in T_z (in $O(\log n)$ time, as described above). Denote by U_i (resp., U_j) the unfolding transformation that maps the points of f_i (resp., f_j) into the plane of f_k. Then $U(q) = U_j^{-1} U_i(q)$.

We describe next the computation of U_i, and U_j is computed analogously. If f_i equals f_k, then U_i is the identity transformation. Otherwise, denote by ν_i the leaf of T_z that stores the polytope edge $f_i \cap f_{i+1}$, and denote by r the root of T_z. We traverse, bottom up, the path $\mathcal{P}$ from ν_i to r, and compose the transformations stored at the nodes of $\mathcal{P}$, initializing U_i as the identity transformation and proceeding as follows. We define a node ν of $\mathcal{P}$ to be a *left turn* (resp., *right turn*) if we reach ν from its left (resp., right) child and proceed to its parent ν' so that ν is the right (resp., left) child of ν'. When we reach a left (resp., right) turn ν that stores U_ν, we update $U_i := U_\nu U_i$ (resp., $U_i := U_\nu^{-1} U_i$). If we reach r from its right child, we do nothing; otherwise we update $U_i := U_r U_i$, where U_r is the transformation stored at r. See Figure 14 for an illustration.

Thus, U_i (and U_j) can be computed in $O(\log n)$ time, and so $U(q) = U_j^{-1} U_i(q)$ can be computed in $O(\log n)$ time. (It is possible to slightly optimize the procedure by computing $U(q)$ directly without the explicit computation of U_i, U_j; that is, $U(q)$ can be computed by traversing the paths from ν_i, ν_j up to their common ancestor instead of traversing all the way to r. However, this optimization does not speed up our algorithm asymptotically.) Handling analogous queries at some fixed x or y is done similarly, using the trees T_x or T_y, respectively.

Hence we can compute, in $O(\log n)$ time, the image of any point $q \in \partial P$ in any unfolding

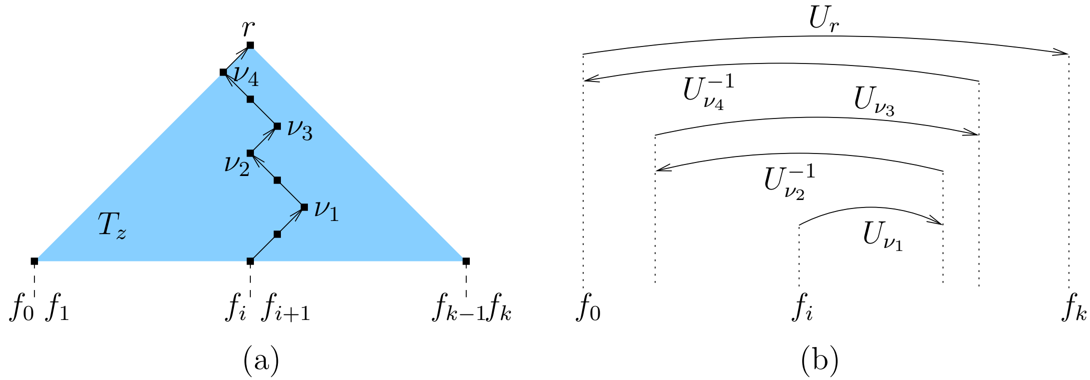

Figure 14: *Constructing U_i by traversing the path from the polytope edge succeeding the facet f_i to the root r of T_z. (a) The nodes ν_1, ν_3 are the left turns, and the nodes ν_2, ν_4 are the right turns in this example. (b) Composing the corresponding transformations stored at $\nu_1, \ldots, \nu_4$ and at r.*

formed by a contiguous sequence of polytope edges crossed by an axis-parallel plane that intersects the facet of q. The surface unfolding data structure that answers these queries requires $O(n \log n)$ space and $O(n \log n)$ preprocessing time.

Lemma 2.11. *Given the 3D-subdivision S_{3D}, the conforming surface subdivision S can be constructed in $O(n \log n)$ time and space.*

Proof: First, we construct the surface unfolding data structure (the enhanced persistent trees T_x, T_y, and T_z) in $O(n \log n)$ time, as described above. Then, we use this data structure to compute the endpoints of all the transparent edges in $O(n \log n)$ time, as follows.

For each subface h of S_{3D}, we use the data structure to find $P \cap h$ in $O(\log n)$ time. If $P \cap h$ is a single component, we split it at its rightmost and leftmost points into two portions as described in the beginning of Section 2.3 — it takes $O(\log n)$ time to locate the split points using a binary search.

To split the intersecting transparent edges, we check each pair of such edges (e, e') that might intersect, as follows. First, we find, in the surface unfolding data structure, the edge sequences $\mathcal{E}$ and $\mathcal{E}'$ traversed by e and e', respectively (by locating the cross sections $P \cap h, P \cap h'$, where h, h' are the respective subfaces of S_{3D} that induce e, e'). Denote by $\mathcal{F} = (f_0, \ldots, f_k)$ (resp., $\mathcal{F}' = (f'_0, \ldots, f'_{k'})$) the corresponding facet sequence of $\mathcal{E}$ (resp., $\mathcal{E}'$). We search for f_0 in $\mathcal{F}'$, using the unfolding data structure. If it is found, that is, both e and e' intersect f_0, we unfold both edges to the plane of f_0 and check whether they intersect each other within f_0. We search in the same manner for f_k in $\mathcal{F}'$, and for f'_0 and $f'_{k'}$ in $\mathcal{F}$. This yields up to four possible intersections between e and e' (if all searches fail, e does not cross e'), by Lemma 2.5. Each of these steps takes $O(\log n)$ time. As follows from Lemma 2.4, there are only $O(n)$ candidate pairs of transparent edges, which can be found in a total of $O(n)$ time; hence the whole process of splitting transparent edges takes $O(n \log n)$ time.

Once the transparent edges are split, we combine their pieces to form the boundary cycles of the cells of the surface subdivision. This can easily be done in time $O(n)$.

The optimization that deletes each group of surface cells whose union completely covers

exactly one hole of a single surface cell and contains no vertices of P also takes $O(n)$ time (using, e.g., DFS on the adjacency graph of the surface cells), since, during the computation of the cell boundaries, we have all the needed information to find the transparent edges to be deleted.

Hence, all the steps in the construction of S take a total of $O(n \log n)$ time, and this concludes the proof of the lemma. $\Box$

This completes the description and analysis of the conforming surface subdivision. The shortest path algorithm, described in Section 4, heavily relies on the well-covering property of this subdivision. But first, in Section 3, we establish some key geometric properties of shortest paths and define data structures for surface unfoldings, which are needed for our algorithm.

3 Surface Unfoldings and Shortest Paths

In this section we derive several properties of the surface subdivision S, and use them to design procedures that unfold ∂P efficiently, which will be required by the shortest path algorithm described in Sections 4 and 5. In the process, we show how to represent the unfolded regions of ∂P used in our shortest path algorithm as *Riemann structures* (defined in detail later in this section). Informally, this representation consists of planar "flaps", all lying in a common plane of unfolding, that are locally glued together without overlapping, but may globally have some overlaps, which however are ignored, since we consider the corresponding flaps to lie at different "layers" of the unfolding. The basic units of this structure are the *building blocks* (the "flaps") defined in Section 3.1.

3.1 Building blocks and contact intervals

Maximal connecting common subsequences. Let e and e' be two transparent edges, and let $\mathcal{E} = (\chi_1, \chi_2, \ldots, \chi_k)$ and $\mathcal{E}' = (\chi'_1, \chi'_2, \ldots, \chi'_{k'})$ be the respective polytope edge sequences that they cross. We say that a common (contiguous) subsequence $\tilde{\mathcal{E}}$ of $\mathcal{E}$ and $\mathcal{E}'$ is *connecting* if none of its edges $\tilde{\chi}$ is intersected by a transparent edge between $\tilde{\chi} \cap e$ and $\tilde{\chi} \cap e'$; see Figure 15(a). We define $G(e, e')$ to be the collection of all *maximal* connecting common subsequences of $\mathcal{E}$ and $\mathcal{E}'$. The subsequences of $G(e, e')$ do not share any polytope edge.

Let e and $\mathcal{E}$ be as above, and let v be a vertex of P. Denote by $\mathcal{E}' = (\chi'_1, \chi'_2, \ldots, \chi'_{k'})$ the cyclic sequence of polytope edges that are incident to v, in their counterclockwise order about v. We regard $\mathcal{E}'$ as an infinite cyclic sequence, and we define $G(e, v)$ to be the collection of *maximal* connecting common subsequences of $\mathcal{E}$ and $\mathcal{E}'$, similarly to the definition of $G(e, e')$. See Figure 15(b). Here too the elements of $G(e, v)$ are pairwise disjoint.

Remark: Note that if there are two subsequences $\tilde{\mathcal{E}}_1, \tilde{\mathcal{E}}_2$ in $G(e, e')$ or in $G(e, v)$ that are separated because of some transparent edge e'' that violates condition (2) for in-between edges, then e'' must be part of a transparent edge cycle (that "separates" $\tilde{\mathcal{E}}_1$ and $\tilde{\mathcal{E}}_2$) that

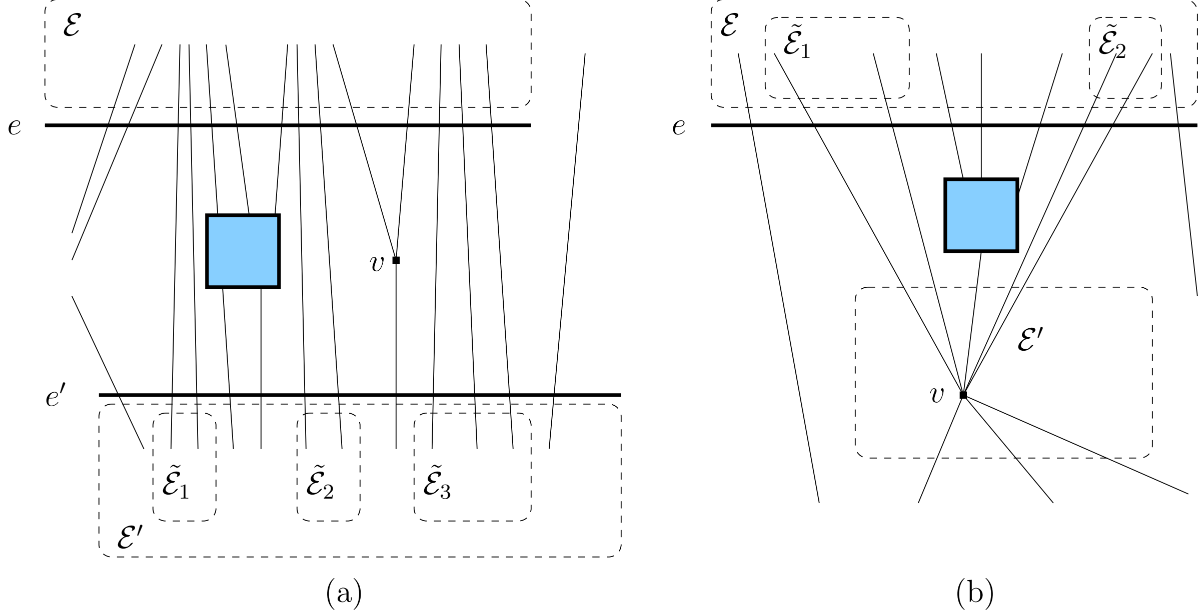

Figure 15: *Maximal connecting common subsequences of polytope edges (drawn as thin solid lines) in (a) $G(e,e')$, and (b) $G(e,v)$. The transparent edges are drawn thick, and the interiors of the transparent boundary edge cycles that separate $\tilde{\mathcal{E}}_1$ and $\tilde{\mathcal{E}}_2$ are shaded.*

contains a vertex of P (the shaded squares illustrated in Figure 15 (a) and (b)), since each group of surface cells whose union completely covers exactly one hole of a single surface cell and contains no vertices of P is deleted during the optimization of S.

We say that each subsequence in $G(e,e')$ *connects* the two transparent edges e, e'. The notion is symmetric (by definition), i.e., $G(e,e') = G(e',e)$. Similarly, each subsequence in $G(e,v)$ is said to *connect* the transparent edge e and the vertex v.

The building blocks. Let c be a cell of the surface subdivision S. Denote by $E(c)$ the set of all the transparent edges on ∂c. Denote by $V(c)$ the set of (zero or one) vertices of P inside c (recall the properties of S). Define $G(c)$ to be the union of all collections $G(x,y)$ so that x, y are distinct elements of $E(c) \cup V(c)$.

Fix such a pair of distinct elements $x, y \in E(c) \cup V(c)$. Let $\mathcal{E}_{x,y} \in G(x,y)$ be a maximal subsequence that connects x and y, and let $\mathcal{F} = (f_0, f_1, \ldots, f_k)$ be its corresponding facet sequence. We define the *shortened facet sequence* of $\mathcal{E}_{x,y}$ to be $\mathcal{F} \setminus \{f_0, f_k\}$ (see Figure 16), and note that the shortened sequence can be empty.

We define the following four types of *building blocks* of a surface cell c.

Type I: Let f be a facet of ∂P that contains at least one endpoint of some transparent edge of ∂c in its closure. Any connected component of the intersection region $c \cap f$ that meets the interior of f and has an endpoint of some transparent edge of ∂c in its closure is a *building block of type I* of c. See Figure 17 for an illustration.

Type II: Let v be the unique vertex in $V(c)$ (assuming it exists), e a transparent edge in

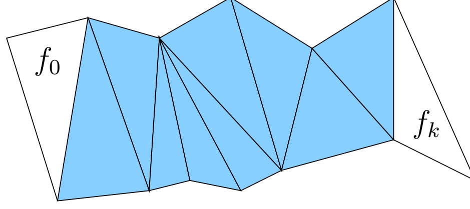

Figure 16: *A facet sequence and its shortened facet sequence (shaded).*

∂c, and $\mathcal{E}_{e,v} \in G(e,v)$ a maximal connecting subsequence between e and v. Then the region B, between e and v in the *shortened* facet sequence of $\mathcal{E}_{e,v}$, if nonempty, is a *building block of type II* of c. More precisely, B is the union, over all facets f in the shortened facet sequence, of the portion of f between $e \cap f$ and v (which is a vertex of f). We say that the maximal connecting edge sequence $\mathcal{E}_{e,v}$ *defines* B. See Figure 18(a) for an illustration of an unfolded building block of type II (the unfolding of a building block is defined below).

Type III: Let e, e' be two distinct transparent edges in ∂c, and let $\mathcal{E}_{e,e'} \in G(c)$ be a maximal connecting subsequence between e and e'. The region B between e and e' in the *shortened* facet sequence of $\mathcal{E}_{e,e'}$, if nonempty, is a *building block of type III* of c. (Again, this can be defined more precisely as in the case of type II blocks.) We say that the maximal connecting edge sequence $\mathcal{E}_{e,e'}$ *defines* B (although in this case the sequence alone does not define B uniquely; for this e and e' must also be specified). See Figure 18(b) for an illustration of an unfolded building block of type III.

Type IV: Let f be a facet of ∂P. Any connected component of the region $c \cap f$ that meets the interior of f, does not contain endpoints of any transparent edge, and whose boundary contains a portion of each of the *three* edges of f, is a *building block of type IV* of c. See Figure 19 for an illustration. (Note that if the region meets only *two* edges of f then it must be a (portion of a) building block of type II or III.)

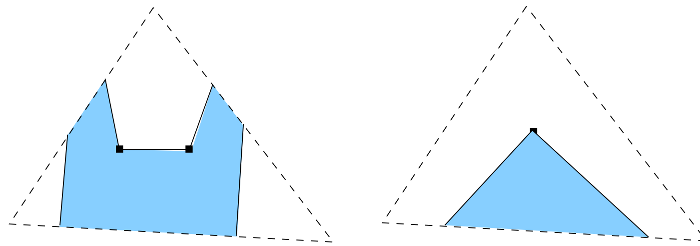

Figure 17: *Two examples of building blocks of type I (shown shaded), each contained in a single facet of ∂P (the dashed triangle).*

We associate with each building block one or two edge sequences along which it can be unfolded. For blocks B contained in a single facet, we associate with B the empty sequence. For other blocks B (which must be of type II or III), the maximal connecting edge sequence $\mathcal{E} = (\chi_1, \ldots, \chi_k)$ that defines B contains at least two polytope edges. Then we associate with B the two *shortened* (possibly empty) sequences $(\chi_2, \ldots, \chi_{k-1})$ and $(\chi_{k-1}, \ldots, \chi_2)$. In either case, none of the sequences associated with B can be cyclic, and if there are two associated sequences $\mathcal{E}_1, \mathcal{E}_2$, then the unfolded images $U_{\mathcal{E}_1}(B), U_{\mathcal{E}_2}(B)$ are congruent.

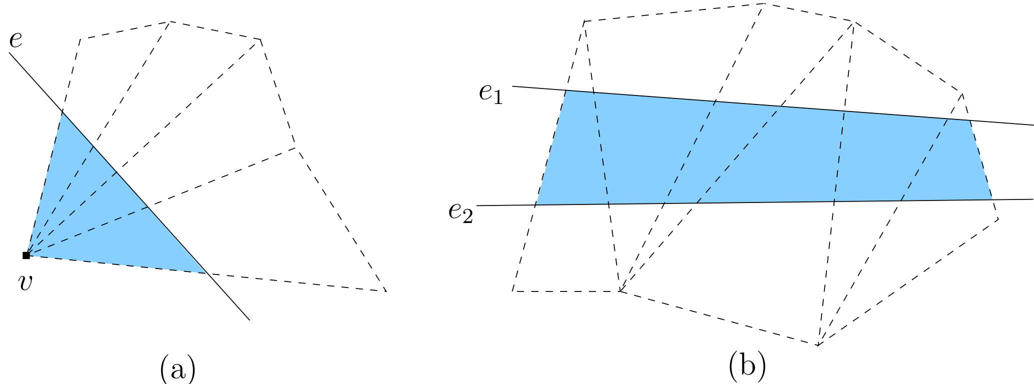

Figure 18: *(a) The unfolding of a building block of type II. (b) The unfolding of a building block of type III. The unfolded block is shaded in both cases.*

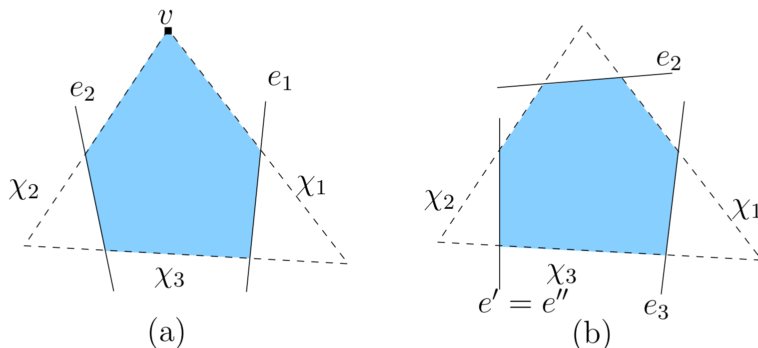

Figure 19: *Two examples of building blocks B of type IV (shown shaded): (a) B is a pentagon containing a vertex v of P. (b) B is a hexagon containing no vertices of P.*

We say that two distinct points $p, q \in \partial P$ *overlap* in the unfolding $U_{\mathcal{E}}$ of some edge sequence $\mathcal{E}$, if $U_{\mathcal{E}}(p) = U_{\mathcal{E}}(q)$. We say that two sets of surface points $X, Y \subset \partial P$ *overlap* in $U_{\mathcal{E}}$, if there are at least two points $x \in X$ and $y \in Y$ so that $U_{\mathcal{E}}(x) = U_{\mathcal{E}}(y)$.

Lemma 3.1. *Let c be a surface cell of S, and let B be a building block of c. Let $\mathcal{E}$ be an edge sequence associated with B. Then no two points $p, q \in B$ overlap in $U_{\mathcal{E}}$.*

Proof: We prove the lemma separately for each type of building block.

For blocks of types I, IV, $U_{\mathcal{E}}(p) = p \neq q = U_{\mathcal{E}}(q)$.

Let B be a block of type II. By definition, B is bounded by a portion of a transparent edge e and two portions of edges of P. By Lemma 2.2, $U_{\mathcal{E}}(e)$ is a straight segment, and obviously the unfolded images of edges of P are straight segments either. Hence, each of the three boundary segments of B can meet any other segment in exactly one point, and therefore $U_{\mathcal{E}}(B)$ is a triangle. It follows that, by linearity of the unfolding transformation, no two points of B overlap in the unfolded image.

Finally, let B be a block of type III. Then B is bounded by a pair of transparent edges e, e' and a pair of polytope edges χ, χ', which are the extreme edges in the connecting subsequence $\mathcal{E}_{e,e'} \in G(c)$. The unfolded image of each of these four boundary portions is a

straight segment; from this and from the linearity of the unfolding transformation follows that $U_{\mathcal{E}}(B)$ may only overlap itself if B is composed of two adjacent regions B_1, B_2, so that B_1 (resp., B_2) is bounded by e and χ (resp., e' and χ'), and $U_{\mathcal{E}}(B_1)$ overlaps $U_{\mathcal{E}}(B_2)$. That is, there is a line $\tilde{b} = B_1 \cap B_2$ that intersects B (connecting $e \cap \chi'$ and $e' \cap \chi$), so that in a close vicinity of $\tilde{b}$, points of B from one side of $\tilde{b}$ overlap, in $U_{\mathcal{E}}$, the points of B from the other side of $\tilde{b}$. However, this contradicts either the fact that the unfolded image of a single facet cannot intersect itself, or the definition of the unfolding of a facet sequence, since B is the shortened facet sequence of $\mathcal{E}_{e,e'}$, and no two subsequent facets of an unfolded facet sequence may overlap each other. □

Lemma 3.2. *Let B be a building block of type IV of a surface cell c, and let f be the facet that contains B. Then either (a) B is a convex pentagon, bounded by portions of the three edges of f, a vertex of f, and portions of two transparent edges, or (b) B is a convex hexagon, whose boundary alternates between portions of the edges of f and portions of transparent edges. In the latter case, B contains no vertices of P (i.e., of f).*

Proof: The boundary of B cannot contain two or three vertices of f, by construction of S.

Suppose first that ∂B contains a vertex v of f, incident to the edges χ_1, χ_2 of f; see Figure 19(a). Transparent edges are not incident to vertices of P, by definition. Hence, there is a boundary segment of B that is a portion of χ_1, delimited by v and by an intersection point of χ_1 with some transparent edge e_1. Similarly, there is a boundary portion of B that is a segment of χ_2, delimited by v and by an intersection point of χ_2 with some transparent edge e_2. Denote the edge of f that is opposite to v by χ_3. By definition of building blocks of type IV, ∂B also contains a portion of χ_3, so that $e_1 \neq e_2$. Transparent edges do not cross, and there are no transparent edge endpoints in B, by definition. Hence e_1 and e_2 intersect χ_3, from which claim (a) follows, as is easily seen.

Suppose then that ∂B does not contain a vertex of f. Let χ_1 be an edge of f. Then there exist two transparent edges e_2, e_3 that intersect χ_1, so that the portion of χ_1 delimited by these two intersection points is a segment of ∂B. See Figure 19(b) for an illustration. There are no transparent edge endpoints in B, by definition, so each of these two transparent edges intersects some other edge of f, different from χ_1. Denote by χ_2 the edge of f intersected by e_2, and by χ_3 the edge of f intersected by e_3. If $\chi_2 = \chi_3$, then one of the edges of f does not contribute to ∂B, contrary to the definition of building blocks of type IV. Therefore $\chi_2 \neq \chi_3$, and since ∂B contains no vertices of f, there is a portion of χ_2, delimited by $e_2 \cap \chi_2$ and by the intersection point with some other transparent edge e', which thus contains a segment of ∂B. Similarly, there is a portion of χ_3, delimited by $e_3 \cap \chi_3$ and by the intersection point with some other transparent edge e''. However, we must have $e' = e''$, for otherwise either e' and e'' intersect, or one of them terminates inside f, both of which cases are impossible. Hence claim (b) holds, and the proof of the lemma is completed. □

Corollary 3.3. *Let B be a building block of type II, III, or IV, and let $\mathcal{E}$ be an edge sequence associated with B. Then $U_{\mathcal{E}}(B)$ is convex.*

Proof: If B is of type II, then $U_{\mathcal{E}}(B)$ is a triangle, by definition and by Lemma 3.1. If B is of type IV, then by Lemma 3.2, $U_{\mathcal{E}}(B)$ is a convex pentagon or hexagon. If B is of type III, then $U_{\mathcal{E}}(B)$ is convex by the proof of Lemma 3.1. □

Corollary 3.4. *There are no holes in building blocks.*

Proof: Immediate for blocks of type II, III, IV, and follows for blocks of type I from the optimization procedure described after the proof of Theorem 2.9. □

Lemma 3.5. *Any surface cell c has only $O(1)$ building blocks.*

Proof: There are $O(1)$ transparent edges in c (by construction of S), and therefore $O(1)$ transparent endpoints, and each endpoint can be incident to at most one building block of c of type I.

There are $O(1)$ transparent edges and at most one vertex of P in c, by construction of S. Therefore there are at most $O(1)$ pairs (e', v) in c so that e' is a transparent edge and v is a polytope vertex. Since there are at most $O(1)$ transparent edge cycles in ∂c (that do not include e') that intersect polytope edges delimited by v and crossed by e', and since each such cycle can split the connecting sequence of polytope edges between e' and v at most once, there are at most $O(1)$ maximal connecting common subsequences in $G(e', v)$. Hence, there are $O(1)$ building blocks of type II of c.

Similarly, there are $O(1)$ pairs of transparent edges (e', e'') in c. There are at most $O(1)$ other transparent edges and at most one vertex of P in c that can lie between e' and e'', resulting in at most $O(1)$ maximal connecting common subsequences in $G(e', e'')$. Hence, there are $O(1)$ building blocks of type III of c.

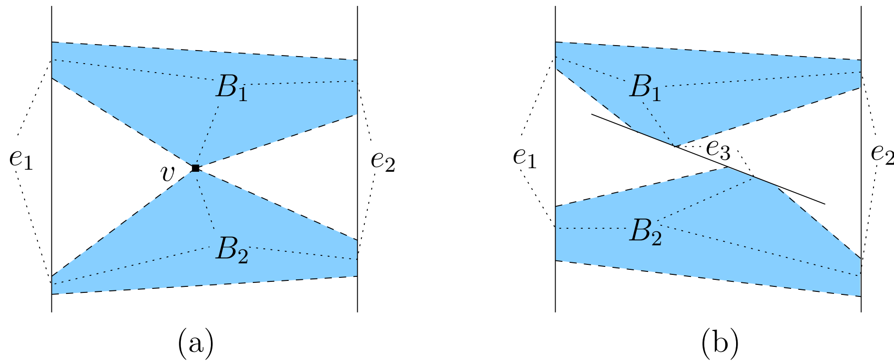

Figure 20: *The triple, of (a) two transparent edges and a vertex of P, or (b) three transparent edges, contributes to two building blocks B_1, B_2. The corresponding graphs $K_{3,2}$ are illustrated by dotted lines. If the triple contributed to three building blocks, we would have obtained an impossible plane drawing of $K_{3,3}$.*

By Lemma 3.2, the boundary of a building block B of type IV contains either two transparent edge segments and a polytope vertex or three transparent edge segments. In either case, we say that this *triple* of elements (either two transparent edges and a vertex of P, or three transparent edges) *contributes* to B. We claim that one triple can contribute to at most two building blocks of type IV (see Figure 20). Indeed, if a triple, say, (e_1, e_2, e_3), contributed to three type IV blocks B_1, B_2, B_3, we could construct from this configuration a plane drawing of the graph $K_{3,3}$ (as is implied in Figure 20), which is impossible. There are $O(1)$ transparent edges and at most one vertex of P in c, by construction of S; therefore

there are at most $O(1)$ triples that contribute to at most $O(1)$ building blocks of type IV of c. $\square$

Lemma 3.6. *The interiors of the building blocks of a surface cell c are pairwise disjoint.*

Proof: Observe that the polytope edges subdivide c into pairwise disjoint components (each contained in a single facet of P). Each building block of type I or IV contains (and coincides with) exactly one such component, by definition. Each building block of type II or III contains one or more such components, and each component is fully contained in the block. Hence it suffices to show that no two distinct blocks can share a component.

If B_1, B_2 are both of type I, then either they lie in different facets, or they are different connected components of c within the same facet, by definition. If B_1 is of type I and B_2 is of another type, then B_1 is a component that contains at least one transparent edge endpoint, while B_2 contains no components that contain transparent edge endpoints, by construction. (It is here, and in the last paragraph of the proof, that we use the fact that blocks of types II, III are constructed from *shortened* facet sequences.) Hence the claim holds if at least one of B_1, B_2 is of type I.

If B_1, B_2 are both of type II, then denote by $\tilde{\mathcal{E}}_1 = \tilde{\mathcal{E}}_{e_1,v}$ and $\tilde{\mathcal{E}}_2 = \tilde{\mathcal{E}}_{e_2,v}$ the maximal connecting sequences corresponding to B_1 and B_2, respectively (recall that c contains at most one vertex of P). Each component of B_1 is a triangle bounded by e_1, v, and two polytope edges, and similarly for B_2. Hence if B_1, B_2 contained a common component Q, then we must have $e_1 = e_2$. Moreover, in this case the shortened facet sequences of $\tilde{\mathcal{E}}_1$ and $\tilde{\mathcal{E}}_2$ must overlap (at the facet containing Q), contradicting the construction of blocks of type II. Hence the lemma holds if B_1, B_2 are both of type II.

By similar arguments, the lemma holds for B_1, B_2 if they are both of type III or one of them is of type II and the other of type III.

If B_1, B_2 are both of type IV, then, by Lemma 3.2 and by definition, they lie in different facets. If B_1 is of type IV and B_2 is of type II or III, then B_1 is a single component (as defined in the beginning of this proof) that contains segments of three different polytope edges on its boundary, while no component in B_2 has this property, by definition.

This completes the proof of the lemma. $\square$

Let B be a building block of a surface cell c. A *contact interval* of B is a maximal straight segment of ∂B that is incident to one polytope edge $\chi \subset \partial B$ and is not intersected by transparent edges, except at its endpoints. See Figures 17–19 for an illustration (contact intervals are drawn as dashed segments on the boundary of the respective building blocks). Our propagation algorithm considers portions of shortest paths that traverse a surface cell c from one transparent edge bounding c to another such edge. Such a path, if not contained in a single building block, traverses a sequence of such blocks, and crosses from one such block to the next through a common contact interval.

Lemma 3.7. *Let c be a surface cell, and let B be one of its building blocks. Then B has at most $O(1)$ contact intervals. If B is of type II or III, then it has exactly two contact intervals, and if B is of type IV, it has exactly three contact intervals.*

Proof: If B is of type I, then B is a (simply connected) polygon contained in a single facet f, so that every segment of ∂B is either a transparent edge segment or a segment of a polytope edge bounding f (transparent edges cannot overlap polytope edges, by Lemma 2.10). Every transparent edge of c can generate at most one boundary segment of B, since it intersects ∂f at most twice. There are $O(1)$ transparent edges, and at most one vertex of P in c, by construction of S. Since each contact interval of B is bounded either by two transparent edges or by a transparent edge and a vertex of P, it follows that B has at most $O(1)$ contact intervals.

If B is of type II, denote by $\mathcal{E}$ one of the edge sequences associated with B. Then the unfolded region $U_{\mathcal{E}}(B)$ is a triangle bounded by two images of polytope edges and one image of a transparent edge. Hence B has exactly two contact intervals.

If B is of type III, denote by $\mathcal{E}$ one of the edge sequences associated with B. Then the unfolded region $U_{\mathcal{E}}(B)$ is a quadrilateral bounded by two images of polytope edges and two images of transparent edges. Hence B has exactly two contact intervals.

If B is of type IV, then it has three contact intervals by construction. □

Corollary 3.8. *Let $I_1 \neq I_2$ be two contact intervals of any pair of building blocks. Then either I_1 and I_2 are disjoint, or their intersection is a common endpoint.*

Proof: By definition. □

Lemma 3.9. *Let c be a surface cell. Then each point of c that is not incident to a contact interval of any building block of c, is contained in exactly one building block of c.*

Proof: Although the proof is straightforward from the definition of basic blocks, we give it in detail for the sake of completeness. Lemma 3.6 implies that no such point can belong to (the interior of) more than one building block. What the current lemma claims is that the union of the closures of the building blocks covers c. Fix a point $p \in c$, and denote by f the facet that contains p. Denote by Q the connected component of $c \cap f$ that contains p. If Q contains in its closure at least one endpoint of some transparent edge of ∂c, then p is in a building block of type I, by definition.

Otherwise, Q must be a convex polygon, bounded by portions of transparent edges and by portions of edges of f; the boundary edges alternate between transparent edges and polytope edges, with the possible exception of a single pair of consecutive polytope edges that meet at the unique vertex v of f that lies in c. Thus only the following cases are possible:

- (i) Q is a triangle bounded by the two edges χ_1, χ_2 of f that meet at v and by a transparent edge e. See Figure 21(a). The subsequence (χ_1, χ_2) connects e and v, hence p is in a building block of type II (f clearly lies in the shortened facet sequence).
- (ii) Q is a quadrilateral bounded by the two edges χ_1, χ_2 of f and by two transparent edges e_1, e_2. See Figure 21(b). Then (χ_1, χ_2) connects e_1 and e_2, hence p is in a building block of type III (again, f lies in the shortened facet sequence).

(iii) Q is a pentagon bounded by the two edges χ_1, χ_2 of f incident to v, by two transparent edges, and by the third edge χ_3 of f. See Figure 21(c). Then, by definition, p lies in a building block of type IV.

(iv) Q is a hexagon bounded by all three edges of f and by three transparent edges. See Figure 21(d). Again, by definition, p lies in a building block of type IV.

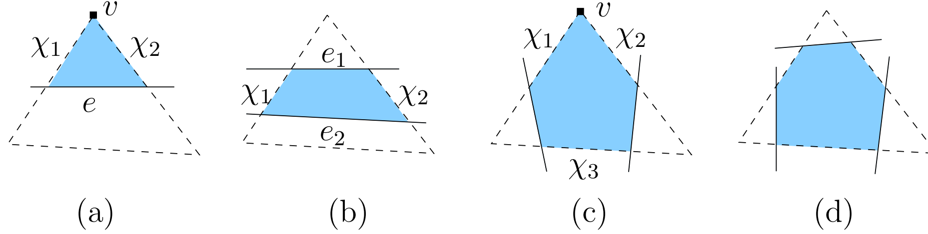

Figure 21: *If Q (shaded) does not contain a transparent endpoint, it must be of one of the depicted forms. In (a) Q is a portion of a building block of type II. In (b), Q is a portion of a building block of type III. In (c) and (d), Q is a building block of type IV.*

This (and the disjointness of building blocks established in Lemma 3.6) completes the proof of the lemma. □

Corollary 3.10. *Let c be a surface cell, and let I be a contact interval of a building block of c. Then there are exactly two different building blocks B_1, B_2 of c so that $I \subset \partial B_1$ and $I \subset \partial B_2$, each on an opposite side of I.*

Proof: Immediate from the preceding analysis. □

The following two auxiliary lemmas (Lemma 3.11 and Lemma 3.12) are used in the proof of Lemma 3.13, which gives an efficient algorithm for computing (the boundaries of) all the building blocks of a single surface cell.

Lemma 3.11. *Let c be a surface cell. We can compute the boundaries of all the building blocks of c of type I in $O(\log n)$ total time.*

Proof: We process one-by-one all the transparent edge endpoints of c. While there is an endpoint a of a transparent edge of c that is not processed yet, do the following. We construct a list L of the vertices of the building block of type I that contains a, and initialize it to $L := (a)$. Denote by f the facet that contains a (which is known from the construction of a), and denote by e one of the transparent edges of ∂c that share a. If e does not intersect ∂f, let b be the second endpoint of e, update $L := L||(b)$ (concatenation of L and (b)), set e to be the other transparent edge of ∂c that is incident to b, and repeat this step until either we find an intersection of e with ∂f or we return back to $b = a$. In the latter case, the whole surface cell c is a single building block of type I (since, by Corollary 3.4, there are no holes inside building blocks). In the former case, denote by x the point of the intersection $e \cap \partial f$. Update $L := L||(x)$. Denote by χ the polytope edge that contains x. Find another transparent edge e' of c that intersects $\chi \cap c$ at a point y closest to x, so that $\overline{xy} \subset c$ (the requirement that we

stay within c defines y uniquely). If there is such a transparent edge, update $L := L||(y)$. Otherwise, χ must lead to the unique vertex v of P inside c. We update $L := L||(v)$, and denote by χ' the other edge of f that is incident to v. Find the transparent edge e' of ∂c that intersects χ' at a point y closest to v. We now continue this tracing procedure from y along e', as above, and continue until we finally get back to a, thereby obtaining the boundary of the type I block containing a.

Since, by Corollary 3.4, there are no holes inside building blocks, after each iteration of the loop we compute one building block of type I of c. Hence, by Lemma 3.5, there are $O(1)$ iterations. In each iteration we process $O(1)$ segments of the current building block boundary. Processing each segment takes $O(\log n)$ time, since it involves $O(1)$ updates of constant-length lists and sets, unfolding $O(1)$ transparent edges and finding $O(1)$ intersections of transparent edges with the facet boundary. (Although we work in a single facet f, each transparent edge that we process is represented relative to its destination plane, which might be incident to another facet of P. Thus we need to unfold it to obtain its portion within f.) Unfolding of a single transparent edge takes $O(\log n)$ time using the surface unfolding data structure (defined in Section 2.4), and computing the intersection point of two straight segments takes $O(1)$ time. Hence the whole procedure takes $O(\log n)$ time. $\square$

Lemma 3.12. *We can compute the boundaries of all the building blocks that are incident to vertices of P in total $O(n \log n)$ time.*

Proof: Let c be a surface cell that contains some (unique) vertex v of P in its interior. Denote by $\mathcal{F}_v$ the cyclic sequence of facets that are incident to v. Compute all the building blocks of type I of c in $O(\log n)$ time, applying the algorithm of Lemma 3.11. Denote by H the set of facets in $\mathcal{F}_v$ that contain building blocks of c of type I *that are incident to* v. Denote by F the set of maximal contiguous subsequences that constitute $\mathcal{F}_v \setminus H$. To compute F, we locate each facet of H in $\mathcal{F}_v$, and then extract the contiguous portions of $\mathcal{F}_v$ between those facets. To traverse $\mathcal{F}_v$ around each vertex v of P takes a total of $O(n)$ time (since we traverse each facet of P exactly three times).

We process F iteratively. Each step picks a nonempty sequence $\mathcal{F} \in F$ and traverses it, until a building block of type II or IV is found and extracted from $\mathcal{F}$.

Let $\mathcal{F}$ be a sequence in F. Since there are no cyclic transparent edges, by construction, $H \cap \mathcal{F}_v \neq \emptyset$, and therefore $\mathcal{F}$ is not cyclic. Denote the facets of $\mathcal{F}$ by $f_1, f_2, \ldots, f_k$, with $k \geq 1$. Denote by $(\chi_1, \chi_2, \ldots, \chi_{k-1})$ the corresponding polytope edge sequence of $\mathcal{F}$ (if $k = 1$, it is an empty sequence). If $k > 1$, denote by χ_0 the edge of f_1 that is incident to v and does not bound f_2, and denote by χ_k the edge of f_k that is incident to v and does not bound f_{k-1}. Otherwise ($k = 1$), denote by χ_0, χ_1 the polytope edges of f_1 that are incident to v. Among all the $O(1)$ transparent edges of ∂c, find the transparent edge e that intersects χ_0 closest to v (by unfolding all these edges and finding their intersections with χ_0). We traverse $\mathcal{F}$ either until it ends, or until we find a facet $f_i \in \mathcal{F}$ so that e intersects χ_{i-1} but does not intersect χ_i (that is, e intersects the polytope edge $\chi \subset \partial f_i$ that is opposite to v).

In the former case (see Figure 22(a)), mark the region of ∂P between e, χ_0, and χ_k as a building block of type II, delete $\mathcal{F}$ from F, and terminate this iteration of the loop. In the latter case, there are two possible cases. If $i > 1$ (see Figure 22(b)), mark the region

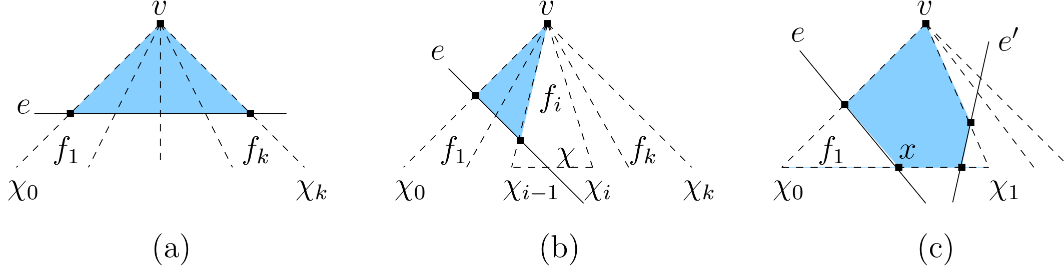

Figure 22: *Transparent edges are thin solid lines, polytope edges are drawn dashed. Extracting from $\mathcal{F}$ building blocks (drawn shaded) of type II (cases (a, b)) or IV (case (c)).*

of ∂P between e, χ_0, and χ_{i-1} as a building block of type II, delete $f_1, f_2, \ldots, f_{i-1}$ from $\mathcal{F}$, and terminate this iteration of the loop. Otherwise ($f_i = f_1$), denote by x the intersection point $e \cap \chi$, and denote by χ' the portion of χ whose endpoint is incident to χ_1. Among all transparent edges of ∂c, find the transparent edge e' that intersects χ' closest to x (such an edge must exist, or else c would contain two vertices of P). The edge e' must intersect χ_1, since otherwise f_i would contain a building block of type I incident to v, and thus would belong to H. See Figure 22(c) for an illustration. Mark the region bounded by $\chi_0, \chi_1, \chi, e, e'$ as a building block of type IV, and delete f_1 from $\mathcal{F}$.

At each iteration we compute a single building block of c, hence there are only $O(1)$ iterations. We traverse the facet sequence around v twice (once to compute F, and once during the extraction of building blocks), which takes $O(n)$ total time for all vertices of P. At each iteration we perform $O(1)$ unfoldings (as well as other constant-time operations), hence the total time of the procedure for all the cells of S is $O(n \log n)$. □

Lemma 3.13. *We can compute (the boundaries of) all the building blocks of all the surface cells of S in total $O(n \log n)$ time.*

Proof: Let c be a surface cell. Compute the boundaries of all the (unfoldings of the) building blocks of c of types I and II, and the building blocks of type IV that contain the single vertex v of P in c, applying the algorithms of Lemmas 3.11 and 3.12. Denote the set of all these building blocks by H. (Note that H cannot be empty, because ∂c contains at least two transparent edges, which have at least two endpoints that are contained in at least one building block of type I.) Construct the list L of the contact intervals of all the building blocks in H. For each contact interval I that appears in L twice, remove both instances of I from L. If L becomes (or was initially) empty, then H contains all the building blocks of c. Otherwise, each interval in L is delimited by two transparent edges, since all building blocks that contain v are in H. Each contact interval in L bounds two building blocks of c, one of which is in H (it is either of type I or contains a vertex of P in its closure), and the other is not in H and is either of type III or a convex hexagon of type IV. The union of all building blocks of c that are not in H consists of several connected components. Since there are no blocks of H among the blocks in a component, neither transparent edges nor polytope edges terminate inside it; therefore such a component is not punctured (by boundary cycles of transparent edges or by a vertex of P), and its boundary alternates between contact intervals in L and portions of transparent edges. For each contact interval I in L, denote by

$limits(I)$ the pair of transparent edges that delimit it. Denote by Y the partition of contact intervals in L into cyclic sequences, so that each sequence bounds a different component, and so that each pair of consecutive intervals in the same sequence are separated by a single transparent edge. By construction, each contact interval in Y appears in a unique cycle. Let $\mathcal{Y} = (I_1, I_2, \ldots, I_k)$ be a cyclic sequence in Y (with $I_{zk+l} = I_l$, for any $l = 1, \ldots, k$ and any $z \in \mathbb{Z}$). Then, for every pair of consecutive intervals $I_j, I_{j+1} \in \mathcal{Y}$, $limits(I_j) \cap limits(I_{j+1})$ is nonempty, and consists of one or two transparent edges (two if the cyclic sequence at hand is a doubleton). Obviously, any cyclic sequence in Y contains two or more contact intervals.

We process Y iteratively. Each step picks a sequence $\mathcal{Y} \in Y$, and, if necessary, splits it into subsequences, each time extracting a single building block of type III or IV, as follows.

If $\mathcal{Y}$ contains exactly two contact intervals, they must bound a single building block of type III, which we can easily compute, and then discard $\mathcal{Y}$. Otherwise, let I_{j-1}, I_j, I_{j+1} be three consecutive contact intervals in $\mathcal{Y}$, and denote by $\chi_{j-1}, \chi_j, \chi_{j+1}$ the (distinct) polytope edges that contain I_{j-1}, I_j and I_{j+1}, respectively. Define the common bounding edge $e_j = limits(I_j) \cap limits(I_{j+1})$ (there is only one such edge, since $|\mathcal{Y}| > 2$), and denote by $\mathcal{E}_j$ the polytope edge sequence intersected by e_j between χ_j and χ_{j+1}, inclusive. Define $\mathcal{E}_{j-1}$ similarly, as the polytope edge sequence traversed by the transparent edge $e_{j-1} = limits(I_{j-1}) \cap limits(I_j)$, delimited by χ_{j-1} and χ_j, inclusive. Without loss of generality, assume that both $\mathcal{E}_{j-1}$ and $\mathcal{E}_j$ are directed from χ_j, to χ_{j-1} and to χ_{j+1}, respectively. See Figure 23. Using binary search, find the last polytope edge in the maximal subsequence $\bar{\mathcal{E}} = \mathcal{E}_{j-1} \cap \mathcal{E}_j$ that starts from χ_j, and denote this edge by χ.

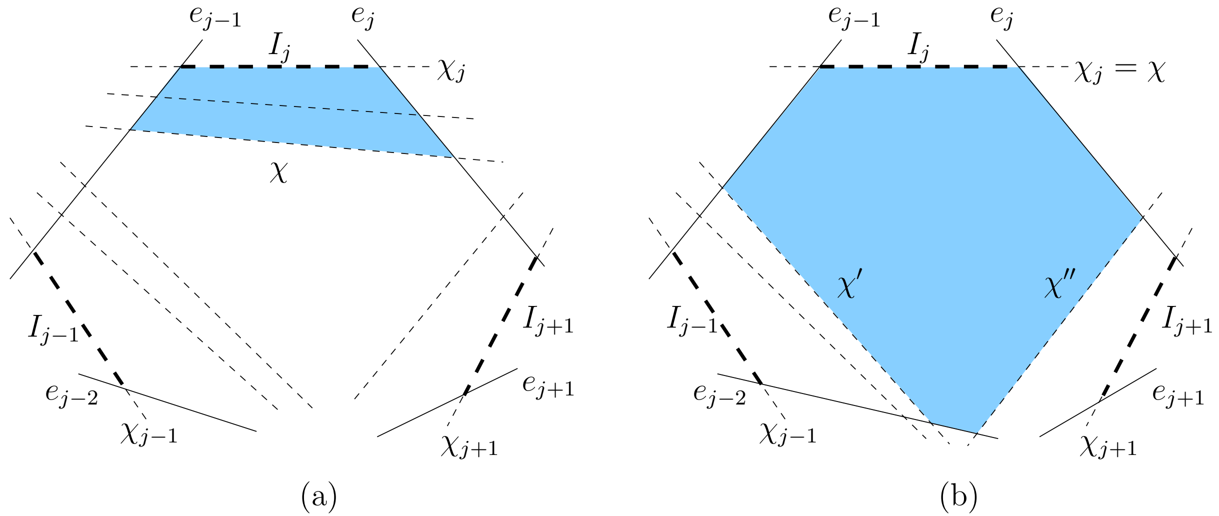

Figure 23: *The unfolded images of transparent edges are solid, while the images of the polytope edges are dashed. The images of the contact intervals are bold dashed segments. There are two possible cases: (a) There is more than one edge in $\bar{\mathcal{E}}$, hence a building block of type III (whose unfolded image is shown shaded) can be extracted. (b) $|\bar{\mathcal{E}}| = 1$ (that is, $\chi_j = \chi$), therefore there must be a building block of type IV (whose image is shown shaded) that can be extracted.*

If $\chi \neq \chi_j$, then we find the unfoldings $U_{\bar{\mathcal{E}}}(e_j)$ and $U_{\bar{\mathcal{E}}}(e_{j-1})$ and compute a new contact interval I'_j, which is the portion of χ bounded by e_j and e_{j-1}. See Figure 23(a). The

quadrilateral bounded by $U_{\bar{\mathcal{E}}}(e_j), U_{\bar{\mathcal{E}}}(e_{j-1}), U_{\bar{\mathcal{E}}}(I'_j)$ and $U_{\bar{\mathcal{E}}}(I_j)$ is the unfolded image of a building block of type III. Delete I_j from $\mathcal{Y}$ and replace it by I'_j.

Otherwise, $\chi = \chi_j$. See Figure 23(b). Denote by χ' the second edge in $\mathcal{E}_{j-1}$, and denote by χ'' the second edge in $\mathcal{E}_j$ (clearly, $\chi' \neq \chi''$). Since all building blocks that contain either a vertex of P or a transparent edge endpoint are in H, the edges χ_j, χ', χ'' bound a single facet, and there is a transparent edge that intersects both χ' and χ'' (otherwise the block of type IV that we are extracting would be bounded by at least four polytope edges — a contradiction). Denote by e the transparent edge that intersects both χ' and χ'' nearest to χ_j or, rather, nearest to e_{j-1} and to e_j, respectively (in Figure 23(b) we have $e = e_{j-2}$). The region bounded by χ_j, χ', χ'' and e_{j-1}, e_j, e is a hexagonal building block of type IV. Compute its two contact intervals that are contained in χ' and χ'', and insert them into $\mathcal{Y}$ instead of I_j. If χ' contains I_{j-1} and χ'' contains I_{j+1}, $\mathcal{Y}$ is exhausted, and we terminate its processing. If χ' contains I_{j-1} and χ'' does not contain I_{j+1}, we remove I_j and I_{j-1} from $\mathcal{Y}$ and replace them by the portion of χ'' between e and e_j. Symmetric actions are taken when χ'' contains I_{j+1} and χ' does not contain I_{j-1}. Finally, if χ' does not contain I_{j-1}, nor does χ'' contain I_{j+1}, we split $\mathcal{Y}$ into two new cyclic subsequences, as shown in Figure 24, and insert them into Y instead of $\mathcal{Y}$.

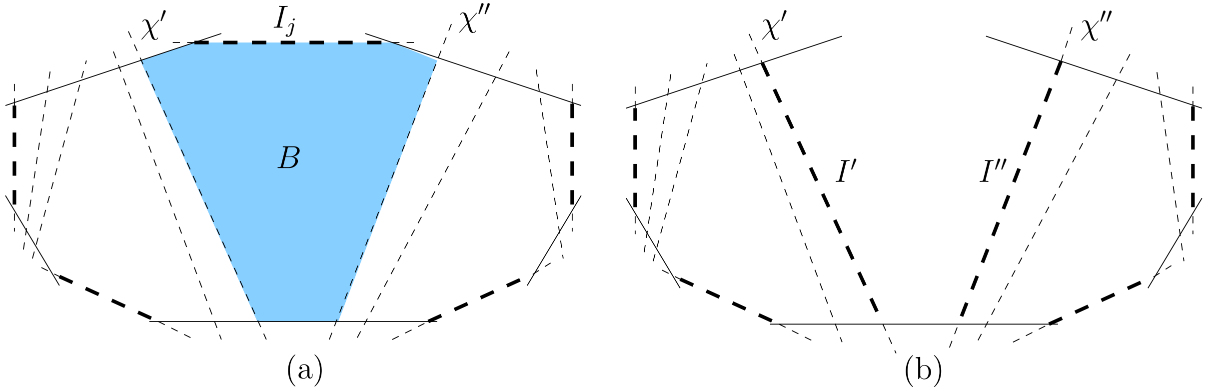

Figure 24: *(a) Before the extraction of the building block B (shaded), the sequence $\mathcal{Y}$ contains five contact intervals (bold dashed segments). (b) After the extraction of B, $\mathcal{Y}$ has been split into two new (cyclic) sequences $\mathcal{Y}', \mathcal{Y}''$ containing the respective contact intervals I', I''. The contact interval I_j is no longer contained in any sequence in Y.*

In each iteration we compute the boundary of a single building block of type III or IV, hence there are $O(1)$ iterations. To compute one building block boundary, we compute $O(1)$ unfoldings, perform $O(1)$ binary searches, and $O(1)$ operations on constant-length lists. Each unfolding calculation and each binary search takes $O(\log n)$ time, hence the time bound follows. $\square$

3.2 Block trees and Riemann structures

In this subsection we combine the building blocks of a single surface cell into more complex structures.

Let e be a transparent edge on the boundary of some surface cell c, and let B be a building block of c so that e appears on its boundary. The *block tree* $T_B(e)$ is a rooted tree whose nodes are building blocks of c, which is defined recursively as follows. The root of $T_B(e)$ is B. Let B' be a node in $T_B(e)$. Then its children are the blocks B'' that satisfy the following conditions.

(1) B' and B'' are adjacent through a common contact interval;

(2) B'' does not appear as a node on the path in $T_B(e)$ from the root to B', except possibly as the root itself (that is, we allow $B'' = B$ if the rest of the conditions are satisfied);

(3) if $B'' = B$, then (a) it is of type II or III (that is, if a root is a building block of type I or IV, it cannot appear as another node of the tree), and (b) it is a leaf of the tree.

Note that a block may appear more than once in $T_B(e)$, but no more than once on each path from the root to a leaf, except possibly for the root B, that may also appear at leaves of $T_B(e)$ if it is of type II or III. However, B cannot appear in any other internal node of the tree. See Figure 25 for an illustration.

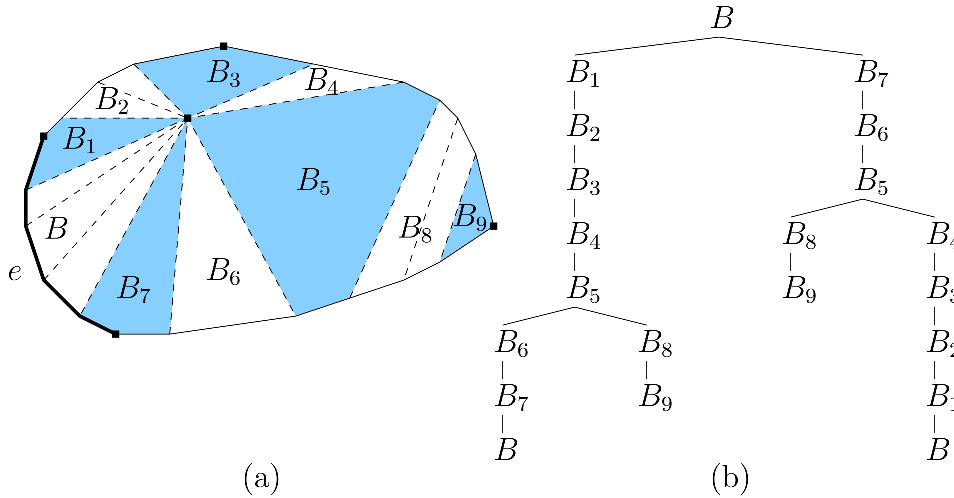

Figure 25: *(a) A surface cell c containing a single vertex of P and bounded by four transparent edges (solid lines) is partitioned in this example into ten building blocks (whose shadings alternate): B_1, B_3, B_7, B_9 are of type I, B, B_2, B_4, B_6 are of type II, B_8 of type III and B_5 of type IV. Adjacent building blocks are separated by contact intervals (dashed lines; other polytope edges are also drawn dashed). (b) The tree $T_B(e)$ of building blocks of c, where e is the (thick) transparent edge that bounds the building block B.*

Remark: The crucial property of $T_B(e)$, as proved in the following lemmas of this subsection, is that each subpath (contained in c) of a shortest path from s to some point in c *that enters c through $B \cap e$ and does not leave c* must traverse a sequence of blocks along some path in $T_B(e)$ (starting at the root). Here is a motivation for the somewhat peculiar way of defining $T_B(e)$ (reflected in properties (2) and (3)). Since each building block is either contained in a single facet (and a single facet is never traversed by a shortest path in more than one

connected segment), or has exactly two contact intervals (and a single contact interval is never crossed by a shortest path more than once), a shortest path $\pi(s,q)$ to a point q in a building block B may traverse B *through its contact intervals* in no more than two connected segments. Moreover, B may be traversed (through its contact intervals) in *two* such segments only if the following conditions hold:

(i) $\pi(s,q)$ must enter B through a point p on a transparent edge on ∂c,

(ii) B consists of components of at least two facets, and p and q are contained in two distinct facets, relatively "far" from each other in B, and

(iii) $\pi(p,q)$ exits B through one contact interval and then re-enters B through another (before reaching q).

See Figure 26 for an illustration. This shows that the initial block B through which a shortest path from s enters a cell c may be traversed a second time, but only if it is of type II or III. After the second time, the path must exit c right away, or end inside B. This explains the way $T_B(e)$ is defined; see Corollary 3.18 and the preceding auxiliary lemmas below for the exact statements and proofs.[8]

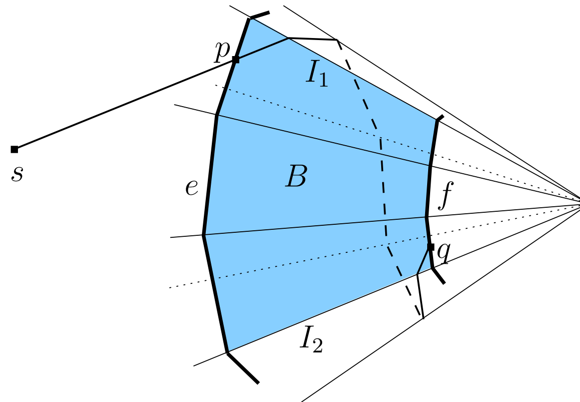

Figure 26: *The shortest path $\pi(s,q)$ enters the (shaded) building block B through the transparent edge e at the point p, leaves B through the contact interval I_1, and then reenters B through the contact interval I_2.*

We denote by $\mathcal{T}(e)$ the set of all block trees $T_B(e)$ of e (constructed from the building blocks of both cells containing e on their boundaries). Note that each block tree in $\mathcal{T}(e)$ contains only building blocks of one cell. We call $\mathcal{T}(e)$ the *Riemann surface structure of* e; it will be used in Section 5 for wavefront propagation block-by-block from e in all directions (this is why we include in it block trees of both surface cells that share e on their boundaries; see Section 5 for details). This structure is indeed similar to standard Riemann surfaces (see, e.g., [44]); its main purpose is to handle effectively (i) the possibility of *overlap* between distinct portions of ∂P when unfolded onto some plane, and (ii) the possibility that shortest

[8]This is not just being too cautious; it is easy to construct concrete examples where such a situation does arise.

paths may traverse a cell c in "homotopically inequivalent" ways (e.g., by going around a vertex or a hole of c in two different ways — see below).

Let us first discuss in some detail the issue of overlaps. We use the Riemann structure in Section 5, in order to be able to propagate wavefronts across cells of S without having to worry about overlaps inside a single block. Without this structure, unfolding an arbitrary portion of ∂P may result in a self-overlapping planar region, making it difficult to apply the propagation algorithm, as described in Sections 4 and 5. It is known that unfolding a convex polytope by cutting it along some of its edges and flattening the boundary of the polytope along the remaining edges may result in a polygonal region that overlaps itself — see [45] for examples, and [34] for a discussion of this topic. However, there exist schemes of cutting a polytope along lines other than its edges, which produce a non-overlapping unfolding — two examples are the *star unfolding* defined in [1] and in [10] (it is called the *outward layout* in [10]; this unfolding is proved to be nonoverlapping in [7]) and the unfolding defined in [40] (called the *input layout* in [10]). It is plausible to conjecture that in the special case of surface cells of S, the unfolding of such a cell does not overlap itself, since S is induced by intersecting ∂P with S_{3D} (which is contained in an arrangement of three sets of parallel planes); O'Rourke proves in [35] that an unfolding of the intersection of a plane with a convex polytope does not overlap itself. This however does not suffice in our case, since (a) we unfold the entire cell, not just planar cross-sections, and (b) the transparent edges that bound our cells differ from these planar cross-sections. A related result [6], which also does not suffice in our case, shows that a specific type of a "band" of the surface of a convex polytope between two parallel planes can be carefully unfolded without overlapping itself. We have not succeeded to prove this conjecture, however, and have overcome this difficulty by employing the above Riemann structure (which also has additional advantages, as discussed later); we leave this conjecture for further research.

Let c be a surface cell. A *block sequence* $\mathcal{B} = (B_1, B_2, \ldots, B_k)$ of c is a sequence of building blocks of c, so that for every pair of consecutive building blocks B_i, B_{i+1} in $\mathcal{B}$, we have $B_i \neq B_{i+1}$, and their boundaries share a common contact interval. We define $\mathcal{E}_{\mathcal{B}}$, the *edge sequence associated with* $\mathcal{B}$, to be the concatenation $\mathcal{E}_1||(\chi_1)||\mathcal{E}_2||(\chi_2)||\ldots||(\chi_{k-1})||\mathcal{E}_k$, where, for each i, χ_i is the polytope edge containing the contact interval that connects B_i with B_{i+1}, and $\mathcal{E}_i$ is the edge sequence associated with B_i that can be extended into $(\chi_{i-1})||\mathcal{E}_i||(\chi_i)$ (recall that there may be two oppositely oriented edge sequences associated with each B_i). Note that, given a sequence $\mathcal{B}$ of at least two blocks, $\mathcal{E}_{\mathcal{B}}$ is unique.

For each block tree $T_B(e)$ in $\mathcal{T}(e)$, each path in $T_B(e)$ defines a block sequence consisting of the blocks stored at its nodes. Conversely, every block sequence of c that consists of *distinct* blocks, with the possible exception of coincidence between its first and last blocks (where this block is of type II or III), appears as the sequence of blocks stored along some path of some block tree in $\mathcal{T}(e)$. We extend these important properties further in the following lemmas.

Lemma 3.14. *Let e, c and B be as above; then $T_B(e)$ has at most $O(1)$ nodes.*

Proof: The construction of $T_B(e)$ is completed, when no path in $T_B(e)$ can be extended without violating conditions (1–3). In particular, each path of $T_B(e)$ consists of distinct

blocks (except possibly for its leaf). Each building block of c contains at most $O(1)$ contact intervals and $O(1)$ transparent edge segments in its boundary, hence the degree of every node in $T_B(e)$ is $O(1)$. There are $O(1)$ building blocks of c, by Lemma 3.5, and this completes the proof of the lemma. □

Remark: Although the proof suggests a possibly large (constant) bound, block trees tend to be rather degenerate, since there is a small number of possible "homotopically inequivalent" ways to traverse c from B (i.e., around at most one vertex of P and a small number of — usually at most one — disjoint inner cycles of ∂c) and therefore each building block of c usually appears only a small number of times in $T_B(e)$; Figure 25 is a good example of this phenomenon, even for a fairly complex cell c.

Note that Lemma 3.14 implies that each building block is stored in at most $O(1)$ nodes of the block tree $T_B(e)$.

Lemma 3.15. *Let e, c and B be as above. Then each building block of c is stored in at least one node of $T_B(e)$.*

Proof: Let $B' \neq B$ be a building block of c. By Lemma 3.9, the union of building blocks covers c, and c is connected, by construction. Moreover, since we assume general position, the relative interior if c is also connected, and the unique vertex v of P inside c (if there is one) also lies in the interior of c.

Hence we can connect a point in the relative interior of B to a point in the relative interior of B' by a path π that lies fully in the relative interior of c and avoids the vertex v.

Consider the sequence $\mathcal{B}_\pi$ of building blocks that π traverses. We may assume that $\mathcal{B}_\pi$ is finite and contains no repetition: If π visits a block B'' twice, we can shortcut the portion of π between these two appearances of B'', exploiting the fact that B'' is connected.

Since π lies fully in the interior of c, it avoids all transparent edges, and thus it must cross between any pair of adjacent blocks in $\mathcal{B}_\pi$ through (the relative interior of) a contact interval. Hence, by construction, $\mathcal{B}_\pi$ appears along some path of $T_B(e)$, as asserted. □

The following two lemmas together summarize the discussion and justify the use of our block trees. (Lemma 3.16 establishes rigorously the informal argument, given right after the block tree definition.)

Lemma 3.16. *Let B be a building block of a surface cell c, and let $\mathcal{E}$ be an edge sequence associated with B. Let p, q be two points in c, so that there exists a shortest path $\pi(p,q)$ that is contained in c and crosses ∂B in at least two different points. Then $U_{\mathcal{E}}(\pi(p,q) \cap B)$ consists of either one or two disjoint straight segments, and the latter case is only possible if p, q lie in B.*

Proof: Since $\pi(p,q)$ is a shortest path, every connected portion of $U_{\mathcal{E}}(\pi(p,q) \cap B)$ is a straight segment.

Suppose first that $p, q \in B$, and assume to the contrary that $U_{\mathcal{E}}(\pi(p,q) \cap B)$ consists of three or more distinct segments (the assumption in the lemma excludes the case of a single segment). Then at least one of these segments is bounded by two points $x, y \in \partial B$

and is incident to neither p nor q. Neither x nor y is incident to a transparent edge, since $\pi(p,q) \subset c$. Hence x and y are incident to two different respective contact intervals I_x and I_y on ∂B. The segment of $U_{\mathcal{E}}(\pi(p,q) \cap B)$ that is incident to p is also delimited by a point of intersection with a contact interval, by similar arguments. Denote this contact interval by I_p, and define I_q similarly. Obviously, the contact intervals I_x, I_y, I_p, I_q are all distinct. Since only building blocks of type I might have four contact intervals on their boundary (by Lemma 3.7), B must be of type I. But then B is contained in a single facet f, and $\pi(p,q)$ must be a straight segment contained in f, and thus cannot cross ∂f at all.

Suppose next that at least one of the points p, q, say p, is outside B. Assume that $U_{\mathcal{E}}(\pi(p,q) \cap B)$ consists of two or more distinct segments. Then at least one of these segments is bounded by two points x, y of ∂B (and is not incident to p). By the same arguments as above, x and y are incident to two different respective contact intervals I_x and I_y. The other segment of $U_{\mathcal{E}}(\pi(p,q) \cap B)$ is delimited by at least one point of intersection with some contact interval I_z, by similar arguments. Obviously, the three contact intervals I_x, I_y, I_z are all distinct. In this case, B is either of type I or of type IV. In the former case, arguing as above, $\pi(p,q) \cap B$ is a single straight segment. In the latter case, B may have three contact intervals, but no straight line can meet all of them. Once again we reach a contradiction, which completes the proof of the lemma. □

Lemma 3.17. *Let e be a transparent edge bounding a surface cell c, and let B be a building block of c so that e appears on its boundary. Then, for each pair of points p, q, so that $p \in e \cap \partial B$ and $q \in c$, if the shortest path $\pi(p,q)$ is contained in c, then $\pi(p,q)$ is contained in the union of building blocks that form a single path in $T_B(e)$ (which starts from the root).*

Proof: Let $p \in e \cap \partial B$ and $q \in c$ be two points as above, and denote by B' the building block that contains q. Denote by $\mathcal{B}$ the building block sequence crossed by $\pi(p,q)$. No building block appears in $\mathcal{B}$ more than once, except possibly B if $B = B'$ (by Lemma 3.16). Hence, the elements of $\mathcal{B}$ form a path in $T_B(e)$ from the root node (that stores B) to a node that stores B', as asserted. □

Corollary 3.18. *Let e be a transparent edge bounding a surface cell c, and let q be a point in c, such that the shortest path $\pi(s,q)$ intersects e, and the portion $\tilde{\pi}(s,q)$ of $\pi(s,q)$ between e and q is contained in c. Then $\tilde{\pi}(s,q)$ is contained in the union of building blocks that define a single path in some tree of $\mathcal{T}(e)$.*

Proof: Let e, c and q be as above. Denote by p the (unique) point $\pi(s,q) \cap e$, and denote by B the building block of c that contains p on its boundary (assuming for the moment that there is only one such block). The portion of $\pi(s,q)$ between p and q is the shortest path from p to q (see Section 2.1), and by Lemma 3.17 it is contained in the union of the building blocks that define a single path in $T_B(e)$. If p lies on a contact interval between two blocks B, B', then, as it is easily verified, $\tilde{\pi}(s,q)$ enters only one of these blocks, and the proof continues as above, using that block. □

Lemma 3.19. *(a) Let e be a transparent edge; then there are only $O(1)$ different paths from a root to a leaf in all trees in $\mathcal{T}(e)$. (b) It takes $O(n \log n)$ total time to construct the Riemann structures $\mathcal{T}(e)$ of all transparent edges e.*

Proof: Let $T_B(e)$ be a block tree in $\mathcal{T}(e)$. There are $O(1)$ different paths from the root node to a leaf of $T_B(e)$ (see the proof of Lemma 3.14). There are two surface cells that bound e, by construction of S, and there are $O(1)$ building blocks of each surface cell, by Lemma 3.5. By Lemma 3.13, we can compute all the boundaries of all the building blocks in overall $O(n \log n)$ time. Hence the claim follows. $\Box$

For the surface cell c that contains s, we similarly define the set of block trees $\mathcal{T}(s)$, so that the root B of each block tree $T_B(s) \in \mathcal{T}(s)$ contains s on its boundary (recall that s is also regarded as a vertex of P). It is easy to see that Corollary 3.18 applies also to the Riemann structure $\mathcal{T}(s)$, in the sense that if q is a point in c, such that the shortest path $\pi(s, q)$ is contained in c, then $\pi(s, q)$ is contained in the union of building blocks that define a single path in some tree of $\mathcal{T}(s)$. It is also easy to see that Lemma 3.19 applies to $\mathcal{T}(s)$ as well.

3.3 Homotopy classes

In this subsection we introduce certain topological constructs that will be used in the analysis of the shortest path algorithm in Sections 4 and 5.

Let R be a region of ∂P. We say that R is *punctured* if either R is not simply connected, so its boundary consists of more than one cycle, or R contains a vertex of P in its interior; in the latter case, we remove any such vertex from R, and regard it as a new artificial singleton hole of R. We call these vertices of P and/or the holes of R *the islands* of R. Let X, Y be two disjoint connected sets of points in such a punctured region R, let $x_1, x_2 \in X$ and $y_1, y_2 \in Y$, and let $\pi(x_1, y_1), \pi(x_2, y_2)$ be two geodesic paths that connect x_1 to y_1 and x_2 to y_2, respectively, inside R. We say that $\pi(x_1, y_1)$ and $\pi(x_2, y_2)$ are *homotopic in R with respect to X and Y* (see [43]), if one path can be continuously deformed into the other within R, while their corresponding endpoints remain in X and Y, respectively. (In particular, none of the deformed paths pass through a vertex of P.) When R is punctured, the geodesic paths that connect, within R, points in X to points in Y, may fall into several different *homotopy classes*, depending on the way in which these paths navigate around the islands of R (see Figure 27 for an illustration). If R is not punctured, all the geodesic paths that connect, within R, points in X to points in Y, fall into a single homotopy class.

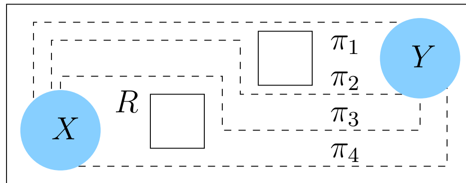

Figure 27: *Two disjoint connected sets of points X, Y (shaded) are contained inside the punctured region R (its boundary consists of three squares). The four paths $\pi_1, \ldots, \pi_4$ fall into three homotopy classes within R with respect to X and Y (π_2 and π_3 are homotopically equivalent).*

In the analysis of the algorithm in Sections 4 and 5, we only encounter homotopy classes of *simple geodesic subpaths* from one transparent edge e to another transparent edge f, inside

a region R that is either a well-covering region of one of these edges or a single surface cell that contains both edges on its boundary. (We call these paths *subpaths*, since the full paths to f start from the source point s.)

Since the algorithm only considers *shortest* paths, we can make the following useful observation. Consider the latter case (where the region R is a surface cell c), and let $\mathcal{B}$ be a path in some block tree $T_B(e)$ within c, that connects e to f. Then all the shortest paths that reach f from e via the building blocks in $\mathcal{B}$ belong to the same homotopy class. Similarly, in the former case (where R is a well-covering region consisting of $O(1)$ surface cells), all the shortest paths that connect e to f via a fixed sequence of building blocks, which itself is necessarily the concatenation of $O(1)$ sequences along paths in separate block trees (joined at points where the paths cross transparent edges between cells), belong to the same homotopy class.

Remark: Note also that, for all subpaths from a transparent edge e to a transparent edge f that belong to a single homotopy class, the polytope edge sequences that they traverse between e and f are all contiguous subsequences of a fixed maximal edge sequence, as is easily verified. This remark is further discussed in the beginning of Section 5.3.1.

4 The Shortest Path Algorithm

This section describes the wavefront propagation phase of the shortest path algorithm. Since this is the core of the algorithm, we present it here in detail, although its high-level description is very similar to the algorithm of [22]. Most of the problem-specific implementation details of the algorithm (which are quite different from those in [22]), as well as the final phase of the preprocessing for shortest path queries, are presented in Section 5.

The algorithm uses the *continuous Dijkstra paradigm*, which simulates a unit-speed *wavefront* expanding from the given source point s, and spreading along the surface of P. However, to ensure efficiency, we do not simulate the *true* wavefront, but an implicit representation thereof, using *one-sided wavefronts*, as detailed below. At *simulation time* t, the true wavefront consists of points whose shortest path distance to s along ∂P is t. The wavefront is a set of closed cycles. Each cycle is a sequence of circular arcs (of equal radii), called *waves*.

Each wave w_i at time t (denoted also as $w_i(t)$) is the locus of endpoints of a collection $\Pi_i(t)$ of shortest paths of length t from s, that satisfy the following condition: There is a fixed polytope edge sequence $\mathcal{E}_i$ crossed by some path $\pi \in \Pi_i(t)$, so that the polytope edge sequence crossed by any other $\pi' \in \Pi_i(t)$ is a prefix of $\mathcal{E}_i$. Denote by $\mathcal{F}_i$ the corresponding facet sequence of $\mathcal{E}_i$. The wave w_i is centered, in the destination plane of $U_{\mathcal{E}_i}$, at the source image $s_i = U_{\mathcal{E}_i}(s)$, called the *generator* of w_i. When w_i reaches, at some time t during the simulation, a point $p \in \partial P$, so that no other wave has reached p prior to time t, we say that s_i *claims* p, and put *claimer*$(p) := s_i$. We say that $\mathcal{E}_i$ is the *maximal polytope edge sequence of s_i at time t*. For each $p \in w_i(t)$ there exists a unique shortest path $\pi(s, p) \in \Pi_i(t)$ that intersects all the edges in the corresponding prefix of $\mathcal{E}_i$, and we denote it as $\pi(s_i, p)$. See Figure 28 for an illustration.

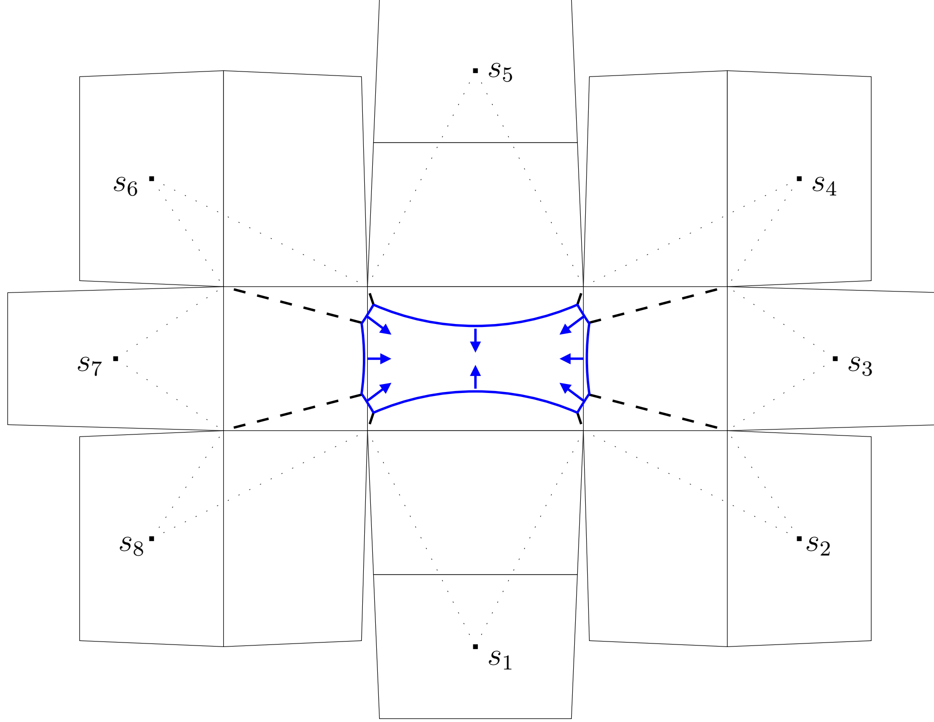

Figure 28: *The true wavefront W (drawn as a cycle of thick circular arcs) at some fixed time t, generated by eight source images $s_1, \ldots, s_8$. The surface of the box P (see the 3D illustration in Figure 29) is unfolded in this illustration onto the plane of the last facet that W reaches; note that some facets of P are unfolded in more than one way (in particular, the facet that contains s is unfolded into eight distinct locations). The dashed lines are the bisectors between the current waves of W, and the dotted lines are the shortest paths to the vertices of P that are already reached by W.*

Note that, left to itself, each point of the wave w_i moves along ∂P in a unique well-defined manner. That is, for any fixed direction from the generator s_i, the (unfolded) wave expands in this direction along a straight ray that traverses a well-defined sequence of polytope edges and facets (unless it reaches a vertex, in which case it stops).

The wave w_i has at most two neighbors w_{i-1}, w_{i+1} in the wavefront, each of which shares a single common point with w_i (if $w_{i-1} = w_{i+1}$, it shares two common points with w_i; for the purpose of the following analysis, these points can be regarded as if shared by w_i with two distinct neighbors $w_{i-1} \neq w_{i+1}$; cf. Figure 29). As t increases and the wavefront expands accordingly (as well as the edge sequences $\mathcal{E}_i$ of its waves), each of the meeting points of w_i with its adjacent waves traces a *bisector*, which is the locus of points equidistant from the generators of the two corresponding waves. The bisector of the two consecutive generators s_i, s_{i+1} in the wavefront is denoted by $b(s_i, s_{i+1})$, and its unfolded image is a straight line; see Section 2.1.1 for details. Figure 29 illustrates the propagation of the true wavefront and the tracing of the bisectors between its waves.

During the wavefront simulation, the combinatorial structure of the wavefront changes at certain *critical events*, which may also change the topology of the wavefront. There are

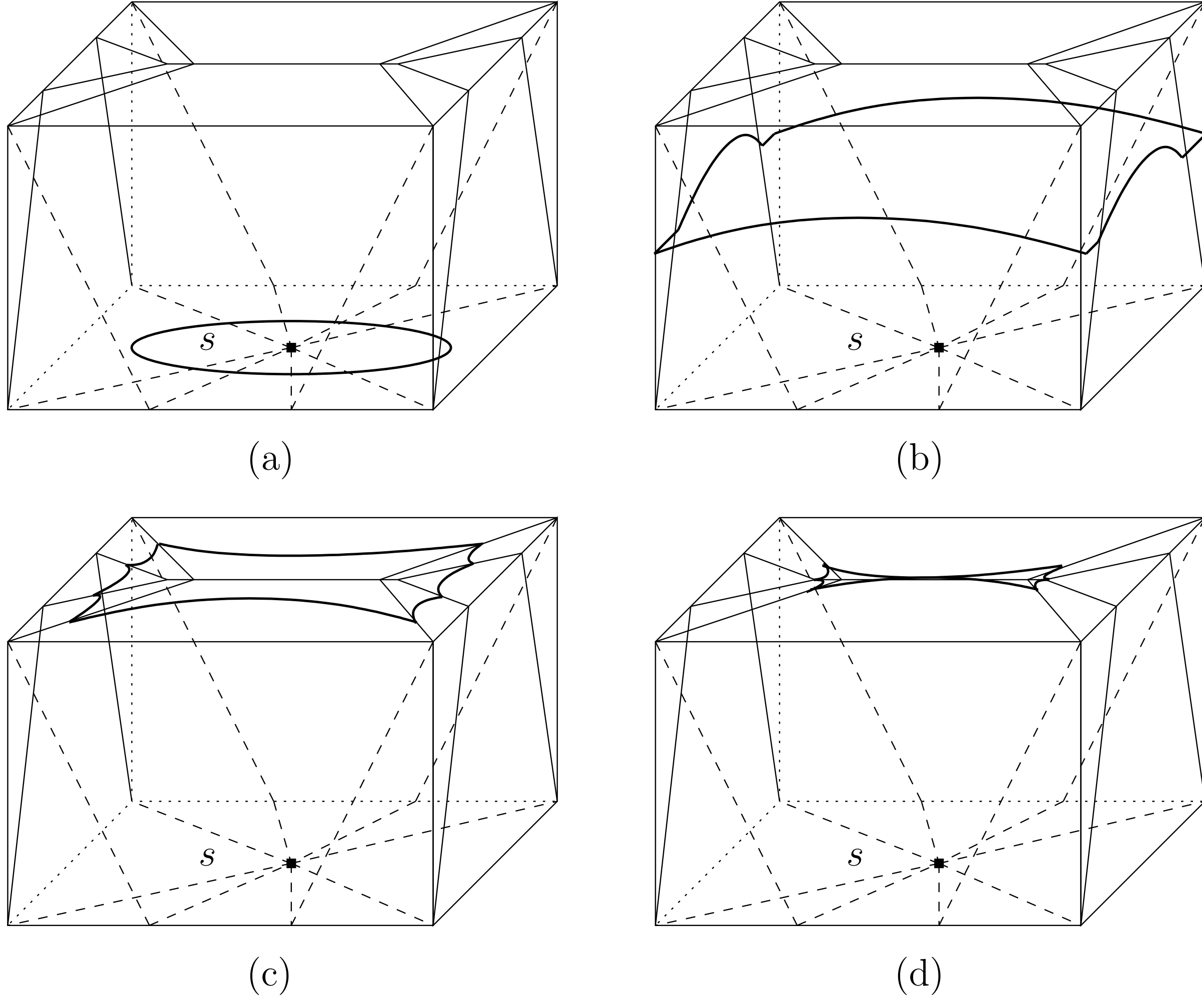

Figure 29: *The true wavefront (drawn as a cycle of thick circular arcs) generated by s at different times t: (a) Before any critical event, the wavefront consists of a single wave. (b) After the first four vertex events the wavefront consists of four (folded) waves. (After the first vertex event, the wavefront has only two waves that meet cyclically and define a single bisector.) (c) After four additional vertex events, the wavefront consists of eight waves. (d) After two additional critical events, that are bisector events, two waves are eliminated. Before the rest of the waves are eliminated, and immediately after (d), the wavefront disconnects into two distinct cycles.*

two kinds of critical events:

(i) *Vertex event*, where the wavefront reaches either a vertex of P or some other boundary vertex (an endpoint of a transparent edge) of the Riemann structure through which we propagate the wavefront. As will be described in Section 5, the wave in the wavefront that reaches a vertex event splits into two new waves after the event. See Figure 30 for an illustration. These are the only events when a new wave is added to the wavefront. Our algorithm detects and processes all vertex events.[9]

[9] A split at a vertex of P is a "real" split, because the two new waves continue past v along two different

(ii) *Bisector event*, when an existing wave is eliminated by other waves — the bisectors of all the involved generators meet at the event point. Our algorithm detects and processes only some of the bisector events, while others are not explicitly detected (recall that we only compute an implicit representation of SPM(s)). See Section 4.3 for further details.

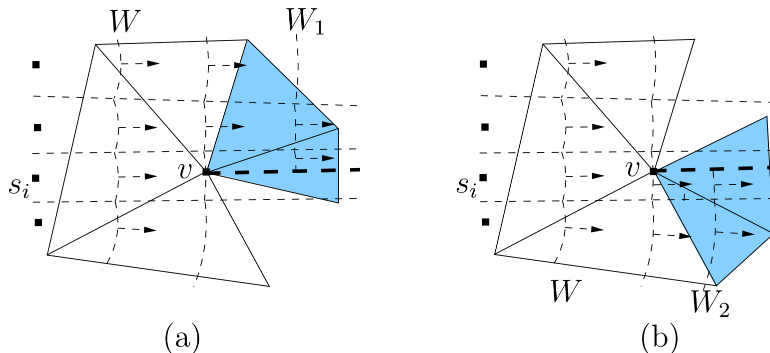

Figure 30: *A vertex event — splitting the wavefront W at the vertex v (the triangles incident to v are unfoldings of its adjacent facets; notice that the sum of all the facet angles at v is less than 2π). The thick dashed line emanating from v is the ray from the source image s_i through v; it replaces the true bisector between the two new wavefronts, which will later be calculated by the merging process. Each of the new wavefronts W_1, W_2 is propagated separately after the event at v (through a different unfolding of the facet sequence around v — see, e.g., the shaded facets, each of which has a different image in (a) and (b)).*

To recap, the real wavefront behaves as described above. However, we simulate a wavefront that may differ from the real wavefront, in the sense that it also includes waves that do not represent shortest paths, because the endpoints of the paths represented by these waves are claimed by other waves. These spurious waves are eliminated in the real wavefront by other waves, at bisector events that we do not detect. Each spurious wave is the locus of endpoints of *geodesic* paths that traverse the same maximal edge sequence, but they need not be shortest paths. Still, our description of bisectors, maximal polytope edge sequences, and critical events that were defined for the true wavefront, also apply to the wavefront propagated by our algorithm.

At each vertex of SPM(s) either a vertex event, or some bisector event (either detected by our algorithm or not) takes place (see Section 2.1.1). However, since the algorithm might propagate more waves than in the true wavefront, some bisector events detected by the algorithm might not be real vertices of SPM(s). See Figure 29 for an illustration of a wavefront, and for the critical events that affect its structure.

The shortest path algorithm has two main phases: a *wavefront propagation phase*, followed by a *map construction phase*. The first phase simulates the wavefront evolution as a function of t, propagating and merging different portions of the wavefront, and determines

edge sequences. A split at a transparent endpoint is an artificial split, used to facilitate the propagation procedure; see Section 5 for details.

the exact locations of some of the real wavefront bisector events (as well as of some "false" bisector events that do not actually occur in SPM(s)). The second phase uses this information to construct an implicit representation of the (true) shortest path map. In the remaining part of this section we describe in detail the first phase, but defer the description of the data structures and implementation details, as well as its detailed complexity analysis, to Section 5. The second phase is also described mostly in Section 5.

Remark: The actual implementation of the propagation phase in Section 5 is slightly different from the high-level description given in this section, although it produces the same output.

4.1 The propagation algorithm

One-sided wavefronts. The algorithm propagates the wavefront through the cells of the conforming surface subdivision S. The wavefront propagates between transparent edges across cells of S. Propagating the exact wavefront explicitly appears to be inefficient (for reasons explained below), so at each transparent edge e we content ourselves with computing two *one-sided wavefronts*, passing through e in opposite directions; together, these one-sided wavefronts carry all the information needed to compute the exact wavefront at e (however, the one-sided wavefronts generally also carry some superfluous information that is not trivial to remove, which is the reason why we do not explicitly merge them to compute the exact wavefront at e).

In more detail, a one-sided wavefront $W(e)$ associated with a transparent edge e (and a specific side of e, which we ignore in this notation), is a sequence of waves $(w_1, \ldots, w_k)$ generated by the respective source images $s_1, \ldots, s_k$ (all unfolded to a common plane, which is the same plane in which we compute the unfolded image of e) with the following properties:

(i) There exists a pairwise openly disjoint decomposition of e into k nonempty intervals $e_1, \ldots, e_k$, appearing in this order along e.

(ii) For each $i = 1, \ldots, k$, for any point $p \in e_i$, the source image that claims p, among the generators of waves that reach p from the fixed side of e, is s_i.

For a fixed side of e, the corresponding wavefront (implicitly) records the times at which the wavefront reaches the points of e from that side. It is possible that the real shortest paths to those points reach them from the other side of e. This information will be picked up by the other wavefront that reaches e from the opposite side. We can interpret the one-sided wavefronts at an edge e by treating e as two-sided, and by labeling each point p on the edge with the time at which the one-sided wavefront reaches p from that side. The algorithm maintains the following crucial *true distance* invariant (which follows from the definition of one-sided wavefronts; its proof, i.e., the proof that one-sided wavefronts are computed correctly, will be given later in Lemma 4.5).

(TD) For any transparent edge e and any point $p \in e$, the true distance $d_S(s, p)$

is the minimum of the two distances to p from the two source images that claim it in the two respective one-sided wavefronts for the opposite sides of e.

See Figure 31 for an illustration. Note that, for simplicity, in Figure 31(a) only one wavefront approaching e from each side is shown; actually, there may be several (but no more than $O(1)$) wavefronts reaching e from each side, which are then merged together (separately for each side) during the construction of the one-sided wavefronts at e itself, as described later in this section. After both one-sided wavefronts at e are computed, they are propagated further, transparently crossing one another through e.

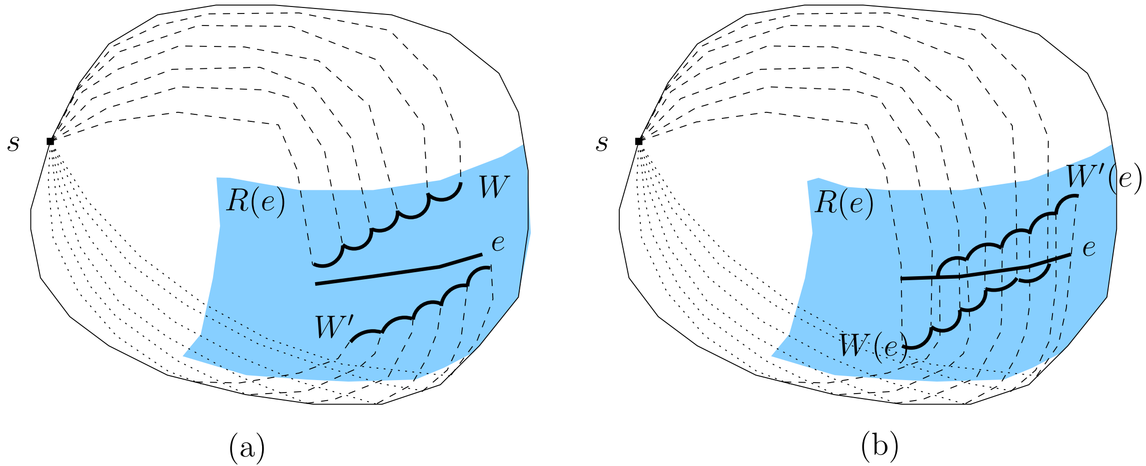

Figure 31: *(a) Two wavefronts W, W' are approaching the transparent edge e from two opposite directions, within the well-covering region $R(e)$ (shaded). (b) Two one-sided wavefronts $W(e), W'(e)$, computed at the simulation time when e is completely covered by W, W', are propagated further within $R(e)$. However, some of the waves that are left in $W(e)$ and $W'(e)$ obviously do not belong to the true wavefront, since there is another wave in the opposite one-sided wavefront that claims the same points of e (before they do).*

Remarks:

(i) We could, in principle, merge the two one-sided wavefronts at e, and the result would yield the *true* shortest path map restricted to e. However, this might take $\Theta(n)$ time per transparent edge e, resulting in overall quadratic algorithm. In contrast, merging wavefronts that reach e from *the same side* can be done more efficiently, as will be shown later.

(ii) As will be shown later, the synchronization mechanism of the algorithm provides an *implicit* interaction between the two opposite one-sided wavefronts on each transparent edge e. Nevertheless, we could have adapted ideas from [22], and allow a limited *explicit* interaction between such wavefronts, discarding some waves that can be ascertained to arrive at e later than some wave from the opposite wavefront. This interaction (that is called "artificial" in [22]) does not affect the asymptotic running time of the algorithm (although it might improve the actual running time, pruning "false" waves closer to the spots where they really disappear from the true wavefront), and makes the algorithm considerably more involved. We therefore ignore this potential enhancement; nevertheless, for the sake of completeness, we provide a detailed description of this artificial interaction in Appendix A.

(iii) Note that a one-sided wavefront $W(e)$ does not represent a fixed time t — each point on e is reached by the corresponding wave at a different time. Additional discussion of this issue will be given later.

The propagation step. The core of the algorithm is a method for computing a one-sided wavefront at an edge e based on the one-sided wavefronts of nearby edges. The set of these edges, denoted *input(e)*, is the set of transparent edges that bound $R(e)$, the well-covering region of e (cf. Section 2.3). To compute a one-sided wavefront at e, we propagate the one-sided wavefronts from each $f \in \mathit{input}(e)$ which has already been processed by the algorithm, to e inside $R(e)$, and then merge the results, separately on each side of e, to get the two one-sided wavefronts that reach e from each of its sides. See Figure 32 for an illustration. The algorithm propagates the wavefronts inside $O(1)$ unfolded images of (portions of) $R(e)$, using the Riemann structure defined in Section 3.2. The wavefronts are propagated only to points that can be connected to the appropriate generator by straight lines inside the appropriate unfolded portion of $R(e)$ (these points are "visible" from the generator); that is, the shortest paths within this unfolded image, traversed by the wavefront as it expands from the unfolded image of $f \in \mathit{input}(e)$ to the image of e, must not bend (cf. Section 2.1 and Section 3). Because the image of the appropriate portion of $R(e)$ needs not be convex, its reflex corners may block portions of wavefronts from some edges of $\mathit{input}(e)$ from reaching e. The paths corresponding to blocked portions of wavefronts that exit $R(e)$ may then re-enter it through other edges of $\mathit{input}(e)$. For any point $p \in e$, the shortest path from s to p passes through some $f \in \mathit{input}(e)$ (unless $s \in R(e)$), so constraining the source wavefronts to reach e directly from an edge in $\mathit{input}(e)$, without leaving $R(e)$, does not lose any essential information.

We denote by $\mathit{output}(e)$ the set of direct "successor" edges to which the one-sided wavefronts of e should be propagated; specifically, $\mathit{output}(e) = \{f \mid e \in \mathit{input}(f)\}$.

The size of (number of edges in) $\mathit{input}(e)$, for any edge e, is constant, by construction. The same holds for $\mathit{output}(e)$:

Lemma 4.1. *For any transparent edge e, $\mathit{output}(e)$ consists of a constant number of edges.*

Proof: Since $|R(f)| = O(1)$ for all f, and each $R(f)$ is a connected set of cells of S, no edge e can belong to $\mathit{input}(f)$ for more than $O(1)$ edges f (there are only $O(1)$ possible connected sets of $O(1)$ cells that contain e on the boundary of their union). □

Remark: Note that, as the algorithm propagates a wavefront from an edge $f \in \mathit{input}(e)$ to e, it may cross other intermediate transparent edges g (see Figure 33). Such an edge g will be processed at an interleaving step, when wavefronts from edges $h \in \mathit{input}(g)$ are propagated to g (and some of the propagated waves may reach g by crossing f first). This "leap-frog" behavior of the algorithm causes some overlap between propagations, but it affects neither the correctness nor the asymptotic efficiency of the algorithm. Moreover, in the actual implementation (see Section 5), propagating from f to e via g will be performed in two (or more) steps, in each of which the propagation is confined to a single cell of S.

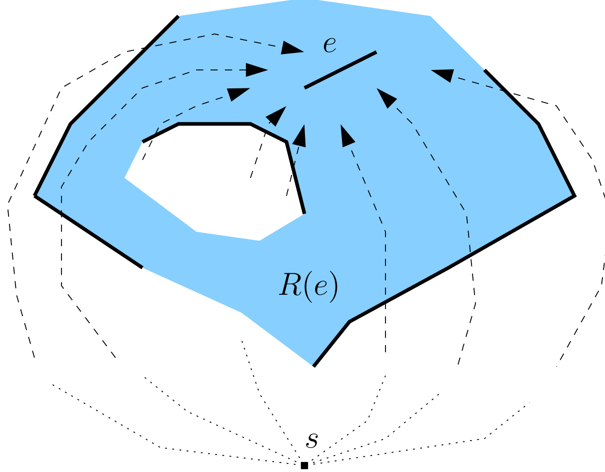

Figure 32: *The well-covering region $R(e)$ is shaded; its boundary consists of two separate cycles. The transparent edge e and all the edges f in input(e) that have been covered by the wavefront before time covertime(e), are drawn as thick lines. The wavefronts $W(f,e)$ that contribute to the one-sided wavefronts at e have been propagated to e before time covertime(e); wavefronts from other edges of input(e) do not reach e either because of visibility constraints or because they are not ascertained to be completely covered at time covertime(e) (in either case they do not include shortest paths from s to any point on e).*

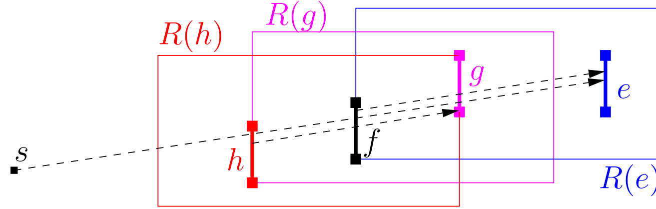

Figure 33: *Interleaving of the well-covering regions. The wavefront propagation from $h \subset \partial R(g)$ to g passes through f, and the propagation from $f \subset \partial R(e)$ to e passes through g.*

The simulation clock. The simulation of the wavefront propagation is loosely synchronized with the real "propagation clock" (that measures the distance from s). The main purpose of the synchronization is to ensure that the only waves that are propagated from a transparent edge e to edges in *output*(e) are those that have reached e no later than $|e|$ simulation time units after e has been completely covered. This, and the well-covering property of e (which guarantees that at this time none of these waves has yet reached any $f \in$ *output*(e)), allow us to propagate further all the shortest paths that cross e by "processing" e only once, thereby making the algorithm adhere to the continuous Dijkstra paradigm, and consequently be efficient. See below for full details.

For a transparent edge e, we define the *control distance from s to e*, denoted by $\tilde{d}_S(s,e)$, as follows. If $s \in R(e)$, and e contains at least one point p that is visible from s within at least one unfolded image $U(R(e))$, for some unfolding U, then e is called *directly reachable* (from s), and $\tilde{d}_S(s,e)$ is defined to be the distance from $U(s)$ to $U(p)$ within $U(R(e))$. The point $p \in e$ can be chosen freely, unless $U(s)$ and $U(e)$ are collinear within $U(R(e))$ — then

p must be taken as the endpoint of e whose unfolded image is closer to $U(s)$. Otherwise ($s \notin R(e)$ or e is completely hidden from s in every unfolded image of $R(e)$), we define $\tilde{d}_S(s,e) = \min\{d_S(s,a), d_S(s,b)\}$, where a, b are the endpoints of e, and $d_S(s,a), d_S(s,b)$ refer to their *exact* values. Thus, $\tilde{d}_S(s,e)$ is a rough estimate of the real distance $d_S(s,e)$, since $d_S(s,e) \le \tilde{d}_S(s,e) < d_S(s,e) + |e|$.[10] The distances $d_S(s,a), d_S(s,b)$ are computed exactly by the algorithm, by computing the distances to a, b within each of the one-sided wavefronts from s to e, and by using the invariant (TD). We compute both one-sided wavefronts for e at the first time we can ascertain that e has been completely covered by wavefronts from either the edges in *input*(e), or directly from s if e is directly reachable. This time is $\tilde{d}_S(s,e) + |e|$, a conservative yet "safe" upper bound of the real time $\max\{d_S(s,q) \mid q \in e\}$ at which e is completely run over by the true (not one-sided) wavefront.

The continuous Dijkstra propagation mechanism computes $\tilde{d}_S(s,e)+|e|$ on the fly for each edge e, using a variable *covertime*(e). Initially, for every directly reachable e, we calculate $\tilde{d}_S(s,e)$, by propagating the wavefront from s within the surface cell which contains s, as described in Section 5, and put *covertime*$(e) := \tilde{d}_S(s,e) + |e|$. For all other edges e, we initialize *covertime*$(e) := +\infty$.

The simulation maintains a time parameter t, called *simulation clock*, which the algorithm strictly increases in discrete steps during execution, and processes each edge e when t reaches the value *covertime*(e). A high-level description of the simulation is as follows:

[10] In fact, if e is not directly reachable, we even have $d_S(s,e) \le \tilde{d}_S(s,e) \le d_S(s,e) + \frac{1}{2}|e|$.

PROPAGATION ALGORITHM

Initialize *covertime*(e), for all transparent edges e, as described above. Store with each directly reachable e, the wavefronts that are propagated to e from s (without crossing edges in *input*(e)).

while there are still *unprocessed* transparent edges **do**

1. **Increase clock:** Select the unprocessed edge e with minimum *covertime*(e), and set $t := covertime(e)$.

2. **Merge:** Compute the one-sided wavefronts for both sides of e, by *merging* together, separately on each side of e, the wavefronts that reach e from that side, either from all the already processed edges $f \in input(e)$ (with $covertime(f) < covertime(e)$, or equivalently, those that have already propagated their wavefronts to e in Step 3 below), or directly from s (those wavefronts are stored at e in the initialization step). Compute $d_S(s, v)$ exactly for each endpoint v of e (the minimum of at most two distances to v provided by the two one-sided wavefronts at e).

3. **Propagate:** For each edge $g \in output(e)$, compute the time $t_{e,g}$ at which one of the one-sided wavefronts from e first reaches an endpoint of g, by *propagating* the relevant one-sided wavefront from e to g. Set $covertime(g) := \min\{covertime(g), t_{e,g} + |g|\}$. Store with g the resulting wavefront propagated from e, to prepare for the later merging step at g.

endwhile

The following lemma establishes the correctness of the algorithm. That is, it shows that *covertime*() is correctly maintained and that the edges required for processing e have already been processed by the time e is processed. The description of Step 2 appears in Section 4.2 as the wavefront *merging* procedure; the computation of $t_{e,g}$ in Step 3 is a byproduct of the propagation algorithm as described below and detailed in Section 5. For the proof of the lemma we assume, for now, that the invariant (TD) is correctly maintained — this crucial invariant will be proved later in Lemma 4.5.

Lemma 4.2. *During the propagation, the following invariants hold for each transparent edge e:*

(a) *The final value of covertime(e) (the time when e is processed) is $\tilde{d}_S(s, e) + |e|$; for directly reachable edges, it is at most $\tilde{d}_S(s, e) + |e|$. The variable covertime(e) is set to this value by the algorithm before or at the time when the simulation clock t reaches this value.*

(b) *The value of covertime(e) is updated only $O(1)$ times before it is set to $\tilde{d}_S(s, e) + |e|$.*

(c) *If there exists a path π from s that belongs to a one-sided wavefront at e, so that a prefix of π belongs to a one-sided wavefront at an edge $f \in input(e)$, then $\tilde{d}_S(s, f) + |f| < \tilde{d}_S(s, e) + |e|$.*

Proof: (a) For directly reachable edges, this holds by definition of the control distance; for other edges e, we prove by induction on the (discrete steps of the) simulation clock, as follows. The shortest path π' to one of the endpoints of e (that reaches e at the time $|\pi'| = t_e = \tilde{d}_S(s,e)$) crosses some $f \in \mathit{input}(e)$ at an earlier time t_f, where $d_S(s,f) \le t_f < \tilde{d}_S(s,f) + |f|$; we may assume that f is the last such edge of $\mathit{input}(e)$. Note that we must have $t_e \ge t_f + d_S(e,f)$. By (W3$_S$), $d_S(e,f) \ge 2\,|f|$, and so $t_e \ge d_S(s,f) + 2\,|f|$. Since $\tilde{d}_S(s,f) < d_S(s,f) + |f|$, we have

$$|\pi'| = t_e \ge d_S(s,f) + 2\,|f| > \tilde{d}_S(s,f) + |f|\,. \tag{1}$$

By induction and by this inequality, f has already been processed before the simulation clock reaches t_e, and so *covertime*(e) is set, in Step 3, to $t_{f,e} + |e| = t_e + |e| = \tilde{d}_S(s,e) + |e|$ (unless it has already been set to this value earlier), at time no later than $t_e = \tilde{d}_S(s,e)$ (and therefore no later than $\tilde{d}_S(s,e) + |e|$, as claimed). By (TD), the variable *covertime*(e) cannot be set later (or earlier) to any *smaller* value; it follows that e is processed at simulation time $\tilde{d}_S(s,e) + |e|$.

(b) The value of *covertime*(e) is updated only when we process an edge f such that $e \in \mathit{output}(f)$ (i.e., $f \in \mathit{input}(e)$), which consists of $O(1)$ edges, by construction.

(c) Any path π that is part of a one-sided wavefront at e must satisfy $d_S(s,e) \le |\pi| < \tilde{d}_S(s,e) + |e|$ (π cannot reach e earlier by definition, and if π reaches e later, then, by (a), e would have been already processed and π would not have contributed to any of the one-sided wavefronts at e). Since π passes through a transparent edge $f \in \mathit{input}(e)$, we can show that $|\pi| > \tilde{d}_S(s,f) + |f|$, by applying arguments similar to those used to derive (1) in (a). Hence we can conclude that $\tilde{d}_S(s,f) + |f| < \tilde{d}_S(s,e) + |e|$. $\square$

Remark: The above synchronization mechanism assures that if a wave w reaches a transparent edge e later than the time at which e has been ascertained to be completely covered by the wavefront, then w will not contribute to either of the two one-sided wavefronts at e. In fact, this important property yields an implicit interaction between the one-sided wavefronts that reach e from the opposite sides, allowing a wave to be propagated further only if it is not too "late"; that is, only if it reaches points on e no later than $2\,|e|$ simulation time units after a wave from the other side of e.

Topologically constrained wavefronts. Let f, e be two transparent edges so that $f \in \mathit{input}(e)$, and let H be a homotopy class of simple geodesic paths connecting f to e within $R(e)$ (recall that when $R(e)$ is punctured, that is, when its boundary is disconnected or when it contains a vertex of P, there might be multiple homotopy classes of that kind; see Section 3.3). We denote by $W_H(f,e)$ the unique maximal (contiguous) portion of the one-sided wavefront $W(f)$ that reaches e by traversing only the subpaths from f to e that belong to the homotopy class H. In Section 5 we regard $W_H(f,e)$ as a "kinetic" structure, consisting of a continuum of "snapshots", each recording the wavefront at some time t. In contrast, in the current section we only consider the (static) resulting wavefront that reaches e, where each point q on (an appropriate portion of) e is claimed by some wave of $W_H(f,e)$, at some time t_q. (Note that this static version is not a snapshot at a fixed time of the kinetic version.) $W_H(f,e)$ is indeed contiguous, since otherwise there must be an island within

the region R_H, which is the locus of all points traversed by all (geodesic) paths in H (see Figure 34) — which contradicts the definition of the homotopy class. We say that $W_H(f,e)$ is a *topologically constrained wavefront* (by the homotopy class H). To simplify notation, we omit H whenever possible, and simply denote the wavefront, somewhat ambiguously, as $W(f,e)$. (The ambiguity is not that bad, though, because there are only $O(1)$ distinct homotopy classes that "connect" f to e.)

A topologically constrained wavefront $W_H(f,e)$ is bounded by a pair of extreme bisectors of an "artificial" nature, defined in one of the two following ways. We say that a vertex of P in $R(e)$ or a transparent endpoint $x \in \partial R(e)$ is a *constraint of* H if x lies on the boundary of R_H; it is easy to see that R_H is bounded by e, f, and by a pair of "chains", each of which connects f with e, and the unfolded image of which (along the polytope edge sequence corresponding to H) is a concave polygonal path that bends only at the constraints of H (this structure is sometimes called an *hourglass*; see [18] for a similar analysis).

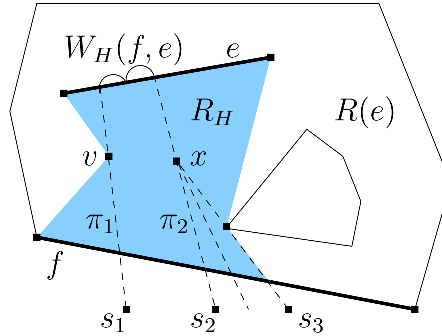

Figure 34: *The "hourglass" region R_H that is traversed by all paths in H is shaded. The extreme artificial bisectors of the topologically constrained wavefront $W_H(f,e)$ are the paths π_1 (from the extreme generator s_1 through the vertex v of P, which is one of the constraints of H) and π_2 (from the generator s_2, that became extreme when its neighbor s_3 was eliminated at a bisector event x, through the location of x).*

Let s' be an extreme generator in $W_H(f,e)$, and let π be a simple geodesic path (in H) from s' that reaches f and touches ∂R_H; see the path π_1 in Figure 34. It is easy to see that if such a path π exists, then it must be an extreme path among all paths encoded in $W_H(f,e)$, since any other path in $W_H(f,e)$ cannot intersect π (see Lemma 4.3 below); we therefore regard π as an extreme artificial bisector of $W_H(f,e)$. Another kind of an extreme artificial bisector arises when, during the propagation of (the kinetic version of) $W_H(f,e)$, an extreme generator s' is eliminated in a bisector event x, as described below, and the neighbor s'' of s' becomes extreme; then the path π from s'' through the location of x becomes extreme in $W_H(f,e)$ — see the path π_2 in Figure 34 (this elimination of extreme generators may occur in succession, affecting in a similar fashion the definition of the extreme artificial bisector).[11] Thus, the type of an extreme bisector of a wavefront can change during the propagation.

When $W_H(f,e)$ is merged, as described in the next subsection, with other topologically constrained wavefronts that reach e, only the artificial bisectors that are also extreme in

[11] Note, however, that, even though π is geodesic, it is not a shortest path to any point beyond x; it is only a convenient (though conservative) way of bounding $W_H(f,e)$ without losing any essential information.

the corresponding resulting one-sided wavefront $W(e)$ "survive" the merging; other artificial bisectors (of generators that are not extreme in $W(e)$) are replaced by true bisectors between the corresponding generators (each of which has been extreme in one of the merged topologically constrained wavefronts, but is not extreme in $W(e)$).

To recap, there are $O(1)$ (topologically constrained) wavefronts that reach e from the left, each of them arrives from one of the $O(1)$ edges $f \in \mathit{input}(e)$, and is constrained by one of the $O(1)$ homotopy classes that connect f to e within $R(e)$. Only those wavefronts that reach e before $\mathit{covertime}(e)$, or equivalently only those that leave edges $f \in \mathit{input}(e)$ satisfying $\mathit{covertime}(f) < \mathit{covertime}(e)$, are propagated to e. After propagating these wavefronts, we *merge* them (at time $\mathit{covertime}(e)$) into a single one-sided wavefront. The same applies to wavefronts that reach e from the right. This merging step is described next.

4.2 Merging wavefronts

Fix a transparent edge e. For specificity, orient e from one endpoint a to its other endpoint b. Consider the computation of the one-sided wavefront $W(e)$ at e that will be propagated further (through e) to, say, the left of e. The *contributing wavefronts* to this computation are all wavefronts $W(f, e)$, for $f \in \mathit{input}(e)$, that contain waves that reach e from the right (not later than at time $\mathit{covertime}(e)$). If e is directly reachable from s, and a wavefront $W(s, e)$ has been propagated from s to the right side of e, then $W(s, e)$ is also contributing to the computation of $W(e)$. The contributing wavefronts for the computation of the one-sided wavefront reaching e from the left are defined symmetrically.

To simplify notation, in the rest of the paper we assume each transparent edge e to be oriented, in an arbitrary direction (unless otherwise specified). For the special case $s \in R(e)$, we also treat the direct wavefront $W(s, e)$ from s to e as if s were another transparent edge f in $\mathit{input}(e)$, unless specifically noted otherwise.

Note that a wavefront $W(f)$ might be split on its way to e into two topologically constrained wavefronts $W_H(f, e), W_{H'}(f, e)$ (by two respective homotopy classes H, H'), each of which may contribute to a *different* one-sided wavefront at e, as illustrated in Figure 35.

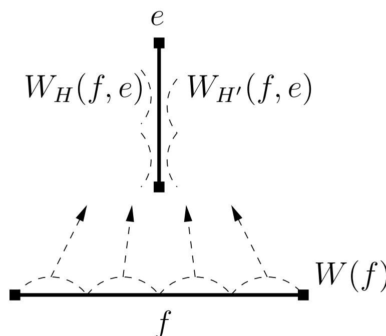

Figure 35: *$W(f)$ contributes to both (opposite) one-sided wavefronts at e.*

The following lemma is a crucial ingredient for the algorithm. It implies that merging wavefronts on the *same side* of a transparent edge e can be done efficiently, in time that

depends on the number of waves that get *eliminated* during the merge.

Lemma 4.3. *Let e be a transparent edge, and let $W(f,e)$ and $W(g,e)$ be two (topologically constrained) contributors to the one-sided wavefront $W(e)$ that reaches e from the right, say. Let x and x' be points on e claimed*[12] *by $W(f,e)$, and let y be a point on e claimed by $W(g,e)$. Then y cannot lie between x and x'.*

Proof: Suppose to the contrary that y does lie between x and x'. Consider a modified environment in which the paths that reach e from the left are "blocked" at e by a thin high obstacle, erected on ∂P at e. This modification does not influence the wavefronts $W(f,e)$ and $W(g,e)$, since no wave reaches e more than once. The simple geodesic paths $\pi(s,x), \pi(s,x')$, and $\pi(s,y)$ in the modified environment connect x and x' to f, and y to g, inside $R(e)$, and lie on the right side of e locally near x, x', and y; see Figure 36(a) for an illustration. By the invariant (TD), the paths $\pi(s,x), \pi(s,x')$, and $\pi(s,y)$ are *shortest paths* from s to these points in the modified environment, and therefore do not cross each other (see Section 2.1). Since $W(f,e), W(g,e)$ are topologically constrained by different homotopies (within $R(e)$), no path traversed by $W(g,e)$ can reach e and be fully contained in the portion Q of ∂P delimited by f, e, and by the portions of $\pi(s,x), \pi(s,x')$ between f and e (shown shaded in Figure 36(a)). Therefore, the portion of the shortest path $\pi(s,y)$ between g and e must enter the region Q through one of the paths $\pi(s,x), \pi(s,x')$, which is a contradiction. Hence y cannot be claimed by $W(g,e)$. $\Box$

Remark: The key property used in the above lemma is that $W(f,e)$ is a topologically constrained wavefront. The lemma may fail if this is not the case. For example, if g is part of an inner cycle of $\partial R(e)$ between f and e, so that (the unconstrained) $W(f,e)$ can bypass g from both sides before it reaches e, then it is possible for $W(g,e)$ to claim an in-between point y on e; see Figure 36(b). Moreover, if $W(g,e)$ reaches e from *the other side* of e then it is possible for $W(g,e)$ to claim portions of $\overline{xx'}$ without claiming x and x'. It is this fact that makes the explicit merging of the two one-sided wavefronts expensive.

Lemma 4.3 is also true if f and g denote two different connected portions of the same transparent edge (a situation that may arise since we topologically constrain the wavefronts). We call the set of all points of e claimed by a contributing wavefront $W(f,e)$ the *claimed portion* or the *claim of* $W(f,e)$. Lemma 4.3 implies that this set is a (possibly empty) *connected* subinterval of e.

We now proceed to describe the *merging process*, applied to the (topologically constrained) contributing wavefronts that claim portions of a transparent edge e (from a fixed side); the process results in the construction of the two one-sided wavefronts at e. As mentioned above, we defer the detailed description of the implementation of the preceding stage, in which the contributing wavefronts are propagated from edges of *input*(e) towards e, to Section 5; nevertheless, some aspects of this propagation, especially the processing of bisector events, are also described, in a higher-level style, in Section 4.3. We assume for now that this propagation has already been correctly executed, so that for each contributing wavefront W, all the critical events involving the generators of W (and no other generators — see

[12]We extend (and relax) the definition of a *claimer* to apply also to contributing wavefronts (previously this term was used only for generators): $W(f,e)$ *claims* a point $p \in e$, if $W(f,e)$ reaches p before any other contributor from the same side of e.

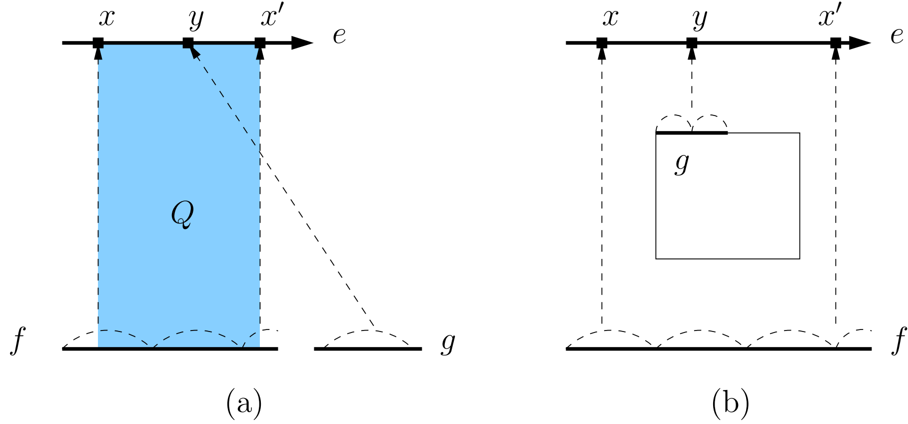

Figure 36: *(a) $W(g,e)$ cannot claim the point y, for otherwise the shortest path $\pi(s,y)$ (that crosses the transparent edge g) would have to cross one of the paths $\pi(s,x), \pi(s,x')$, which is impossible for shortest paths. The region Q delimited by f, e, and the portions of $\pi(s,x), \pi(s,x')$ between f and e is shaded. (b) If $W(f,e)$ is not topologically constrained, $W(g,e)$ may claim an in-between point y on e.*

Section 4.3 for further discussion) during this propagation have already been detected and processed. That is, when starting the merging process at simulation clock $t = covertime(e)$, the correct order of the generators in each contributing wavefront W at time t is known, and the outermost points of e that W reaches (possibly failing to reach the endpoints of e because of possible visibility and/or topological constraints) are also known.

Most of the low-level details of the process are embedded in the various procedures supported by the data structure described in Section 5.1; for now, before proceeding with Lemma 4.4, we briefly review the basic operations that the merging uses, and assert their time complexity bounds. Each contributing wavefront W is maintained as a list of generators in a balanced tree data structure T, where each leaf represents a single generator of W. The unfolding transformation of each generator is also stored in this data structure (in a distributed way, over certain nodes of T), so that we can compute the unfolding of a single source image to a plane traversed by its wave in time proportional to the depth of T. We may therefore assume that each of the operations of constructing a single bisector, finding its intersection point with e, measuring the distance to a point on e from a single generator, and concatenating the lists representing two wavefront portions into a single list, takes $O(\log n)$ time. This will be further explained and verified in Section 5.

Lemma 4.4. *For each transparent edge e and for each $f \in input(e)$, we can compute the claim of each of the wavefront portions $W(f,e)$ that contribute to the one-sided wavefront $W(e)$ that reaches e from the right, say, in $O((1+k)\log n)$ total time, where k is the total number of generators in all wavefronts $W(f,e)$ that are absent from $W(e)$.*

Proof: For each contributing wavefront $W(f,e)$, we show how to determine the claim of $W(f,e)$ in the presence of only one other contributing wavefront $W(g,e)$. The (connected) intersection of these claimed portions, taken over all other $O(1)$ contributors $W(g,e)$, is the part of e claimed by $W(f,e)$ in $W(e)$. This is repeated for each wavefront $W(f,e)$, and

results in the algorithm asserted in the lemma. (Some constant speedup is possible if we merge more than two wavefronts at a time, but we present the process in this way to make the description simpler.)

Orient e from one endpoint a to the other endpoint b. We refer to a (resp., b) as the *left* (resp., *right*) endpoint of e. We determine whether the claim of $W(f,e)$ is *to the left or to the right* of that of $W(g,e)$, as follows. If both $W(f,e)$ and $W(g,e)$ claim a, then, in $O(\log n)$ time, we check which of them reaches it earlier (we only need to check the distances from a to the first and the last generator in each of the two wavefronts, since we assume that $W(f,e)$ and $W(g,e)$ only contain waves that reach e). Otherwise, one of $W(f,e), W(g,e)$ reaches a point $p \in e$ (not necessarily a) that is left of any point reached by the other; by Lemma 4.3, the claim that contains p, by "winning" wavefront, is to the left of the claim of the other wavefront. To find p, we intersect the first and the last (artificial) bisectors of each of $W(f,e), W(g,e)$ with e; p is the intersection closest to a.

A basic operation performed here and later in the merging process is to determine the order of two points x, y along e. To perform this comparison, we compute the polytope edge sequence $\mathcal{E}_e$ crossed by e, and compare $U_{\mathcal{E}_e}(x)$ with $U_{\mathcal{E}_e}(y)$. Using the surface unfolding data structure of Section 2.4, this operation takes $O(\log n)$ time.

Without loss of generality, assume that the claim of $W(f,e)$ is left of that of $W(g,e)$. Note that in this definition we also allow for the case where $W(g,e)$ is completely annihilated by $W(f,e)$.

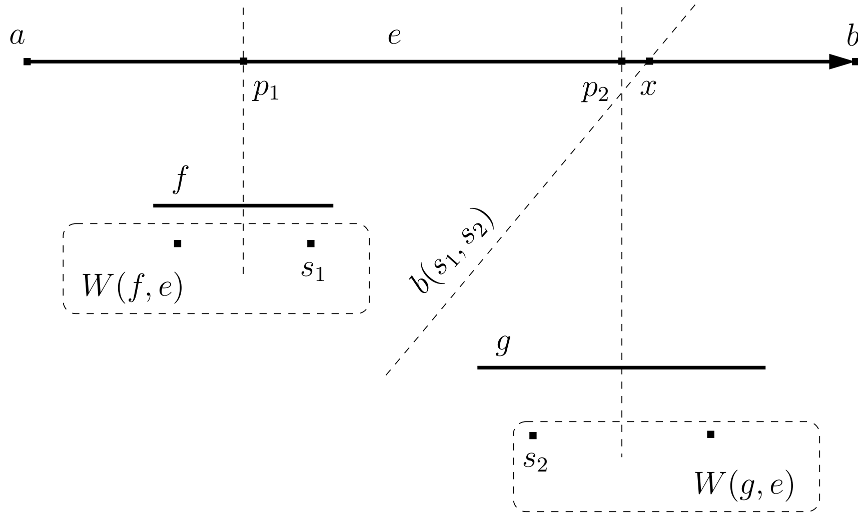

Figure 37: *The source image s_2 is eliminated from $W(e)$, because its contribution to $W(e)$ must be to the left of p_2 and to the right of x, and therefore does not exist along e.*

By Lemma 4.3, we can combine the two wavefronts using only local operations, as follows. Let s_1 denote the generator in $W(f,e)$ that claims the rightmost point on e among all points claimed by $W(f,e)$; by assumption, s_1 is an extreme generator of $W(f,e)$. Let p_1 be the left endpoint of the claim of s_1 on $U_{\mathcal{E}_e}(e)$ (as determined by $W(f,e)$ alone; it is the intersection of $U_{\mathcal{E}_e}(e)$ and the left bisector of s_1). Similarly, let s_2 denote the generator in $W(g,e)$ claiming the leftmost point on e (among all points claimed by $W(g,e)$), and let p_2 be the right endpoint

of the claim of s_2 on $U_{\mathcal{E}_e}(e)$ (as determined by $W(g,e)$ alone). We compute the (unfolded) bisector of s_1 and s_2, and find its intersection point x with $U_{\mathcal{E}_e}(e)$. See Figure 37. If x is to the left of p_1 or x does not exist and the entire e is to the right of $b(s_1, s_2)$, then we DELETE s_1 from $W(f,e)$, reset s_1 to be the next generator in $W(f,e)$, and recompute p_1. If x is to the right of p_2 or x does not exist and the entire e is to the left of $b(s_1, s_2)$, then we update $W(g,e)$, s_2 and p_2 symmetrically. In either case, we recompute x and repeat this test. If p_1 is to the left of p_2 and x lies between them, then x is the right endpoint of the claim of $W(f,e)$ in the presence of $W(g,e)$ and the left endpoint of the claim of $W(g,e)$ in the presence of $W(f,e)$. Again, the correctness of this process follows from the connectedness of the claims of $W(f,e), W(g,e)$ on e.

By combining the claimed portions for all contributors $W(f,e)$, we construct the one-sided wavefront $W(e)$ that reaches e from the right. A fully symmetric procedure is applied to the wavefront that reaches e from the left.

Consider next the time complexity of this merging process. We merge $O(1)$ pairs $W(f,e)$, $W(g,e)$ of wavefronts. Merging a pair of wavefronts involves $O(1+k)$ operations, where k is the number of generators that are deleted from the wavefronts, and where each operation either computes a single bisector, or finds its intersection point with e, or measures the distance to a point on e from a single generator, or deletes an extreme wave from a wavefront, or concatenates two wavefront portions into a single list. As stated above, and detailed in Section 5, each of these basic operations can be implemented in $O(\log n)$ time. Summing over all $O(1)$ pairs $W(f,e)$, $W(g,e)$, the bound follows. $\square$

We defer the few remaining details that allow efficient implementation of the merging procedure to Section 5. The following lemma proves the correctness of the process, with the assumption that the propagation procedure, whose details are not provided yet, is correct. This assumption is ascertained in the rest of the paper.

Lemma 4.5. *(i) Any generator deleted during the construction of a one-sided wavefront at the transparent edge e does not contribute to the* true wavefront *at e. (ii) Assuming that the propagation algorithm deletes a wave from the wavefront not earlier than the time when the wave becomes dominated by its neighbors, every generator that contributes to the true wavefront at e belongs to one of the (merged) one-sided wavefronts at e.*

Proof: The first part is obvious — each point in the claim of each deleted generator s_i along e is reached earlier either by its neighbor generator in the same contributing wavefront or by a generator of a competing wavefront. It is possible that these generators are further dominated by other generators in the true wavefront, but in either case s_i cannot claim any portion of e in the true wavefront. The second part follows by induction on the order in which transparent edges are being processed, based on the following two facts. (i) Any wave that contributes to the true wavefront at e must arrive either directly from s inside $R(e)$, or through some edge f in *input*(e) (by the definition of well-covering). (ii) The one-sided wavefronts at each edge f of *input*(e) that have been covered before e is processed, have already been computed (by Lemma 4.2). Hence each generator s_i that contributes to the true wavefront at e contributes to the true wavefront at some such edge f, and the induction hypothesis implies that s_i belongs to the appropriate one-sided wavefront at f. Since, by the assumption that is ascertained in the next section, the propagation algorithm from f

to e deletes from the wavefront only the waves that become dominated by other waves, s_i participates in the merging process at e, and, by the first part of the lemma, cannot be fully eliminated in that process. □

4.3 The bisector events

When we propagate a one-sided wavefront $W(e)$ to the edges of *output*(e), as will be described in detail in Section 5.2, and when we merge the wavefronts that reach the same transparent edge, as described in Section 4.2, *bisector events* may occur, as defined above. We distinguish between the following two kinds of bisector events.

(i) **Bisector events of the first kind** are detected when we simulate the advance of the wavefront $W(e)$ from e to g to compute the wavefront portion $W(e, g)$, for some $g \in$ *output*(e). In any such event, two non-adjacent generators s_{i-1}, s_{i+1} become adjacent due to the elimination of the intermediate wave generated by s_i; see Figure 38(a) for an illustration. This event is the starting point of $b(s_{i-1}, s_{i+1})$, which reaches g in $W(e, g)$ if both waves survive the trip. Storing and maintaining these events by their "priorities" (distances from s), the algorithm processes all such events that occur before g is ascertained to be covered; that is, before the simulation time *covertime*(g). When such an event occurs, we compute the new bisector $b(s_{i-1}, s_{i+1})$ and delete the eliminated generator from the wavefront. (Further algorithmic aspects of detecting and processing these events are provided later in Section 5.)

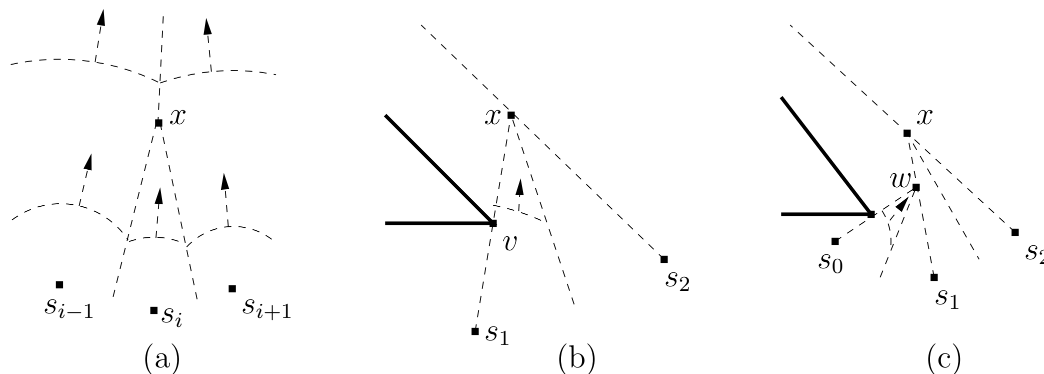

Figure 38: *When a bisector event of the first kind takes place at x, the wave of the corresponding generator is eliminated from the wavefront W: (a) The wave of s_i is eliminated, and the new bisector $b(s_{i-1}, s_{i+1})$ is computed. (b,c) The wave of the leftmost generator s_1 in W is eliminated, and s_2 takes its place; the ray from s_2 through x becomes the leftmost (artificial) bisector of W, instead of the former leftmost bisector, which is either (b) the ray from s_1 through a transparent edge endpoint v (a visibility constraint), or (c) the ray from s_1 through the location w of an earlier bisector event, where s_0, the previous leftmost generator of W, has been eliminated.*

A bisector event, at which the *first* generator s_1 in the propagated wavefront is eliminated, is treated somewhat differently; see Figure 38(b,c) for an illustration. In this case s_1 is deleted from the wavefront W and the next generator s_2 becomes the first in W. The ray from s_2 through the event location becomes the first (that is, extreme), artificial bisector of W, meaning that W needs to be maintained only on the s_2-side of this bisector (which is a

conservative bound). Indeed, any point $p \in \partial P$ for which the path $\pi(s_2, p)$ crosses $b(s_1, s_2)$ into the region of ∂P that is claimed by s_1 (among all generators in W), can be reached by a shorter path from s_1. The case when the *last* generator of W is eliminated is treated symmetrically.

(ii) **Bisector events of the second kind** occur when waves from different topologically constrained wavefronts collide. Our algorithm does not compute these events, but, for the sake of completeness, and for better visualization of the wavefront propagation paradigm, we list them here in some detail.

If a generator s_i contributes to one of the input wavefronts $W(e, g)$ but not to the merged one-sided wavefront $W(g)$ at g, then s_i is involved in at least one bisector event (of the second kind) on the way from e to g, and there must exist some generator s_j in another (topologically constrained) wavefront $W(f, g)$ approaching g, that eliminates the wave of s_i.

See Figure 39 for an illustration of the former case. The exact event location and the bisector $b(s_i, s_j)$ are never explicitly computed by our algorithm. If the claim of s_i on $W(g)$ is *shortened* (but not eliminated) by the wave of a generator s_j of another component wavefront, then s_i is involved in a bisector event that occurs between e and g, and the new bisector $b(s_i, s_j)$ emanates from the event point — this bisector is computed by the algorithm during the merging procedure at g, but the event point itself is not. However, the fact that we do not explicitly compute these bisector event locations will not harm the later stage of preprocessing the resulting structure for shortest path queries (see Section 5.4).

Bisector events of the second kind also occur when two opposite one-sided wavefronts of a transparent edge e collide into each other.

Another case of such an event occurs when a one-sided wavefront $W(e)$ is split during its propagation inside $R(e)$ (either because of a vertex of P or because of a hole of $R(e)$ that may contain one or more vertices of P), and the two portions of the split wavefront partially collide into each other during their further propagation inside $R(e)$, as distinct topologically constrained wavefronts, before they reach $\partial R(e)$ — see Figure 40 for an illustration.

The algorithm never encounters these collisions. It simply propagates each of the wavefronts separately, and then realizes that "something is wrong" — the wavefronts have been propagated into a "loop", namely, into blocks that they have already traversed — and stops the propagation of both wavefronts. See Section 5.3.1 for details.

Tentatively false and true bisector events. Consider the time $t = covertime(e)$ when a transparent edge e is processed and the one-sided wavefronts at e are computed. There may be waves that have reached e before time t (although not earlier than time $t - 2|e|$), and some of these waves could have participated in bisector events of the first kind "beyond" e, which could have taken place before time t. As described in Section 5, the algorithm detects these (currently considered as) "false" bisector events when the wavefronts from the edges in *input*(e) are propagated to e, but the generators that are eliminated in these events are not deleted from their corresponding contributing wavefronts before time t. This is done to ensure that the one-sided wavefronts computed at e correctly represent (together) the true intersection of SPM(s) with e (that is, the invariant (TD) is satisfied); in this sense,

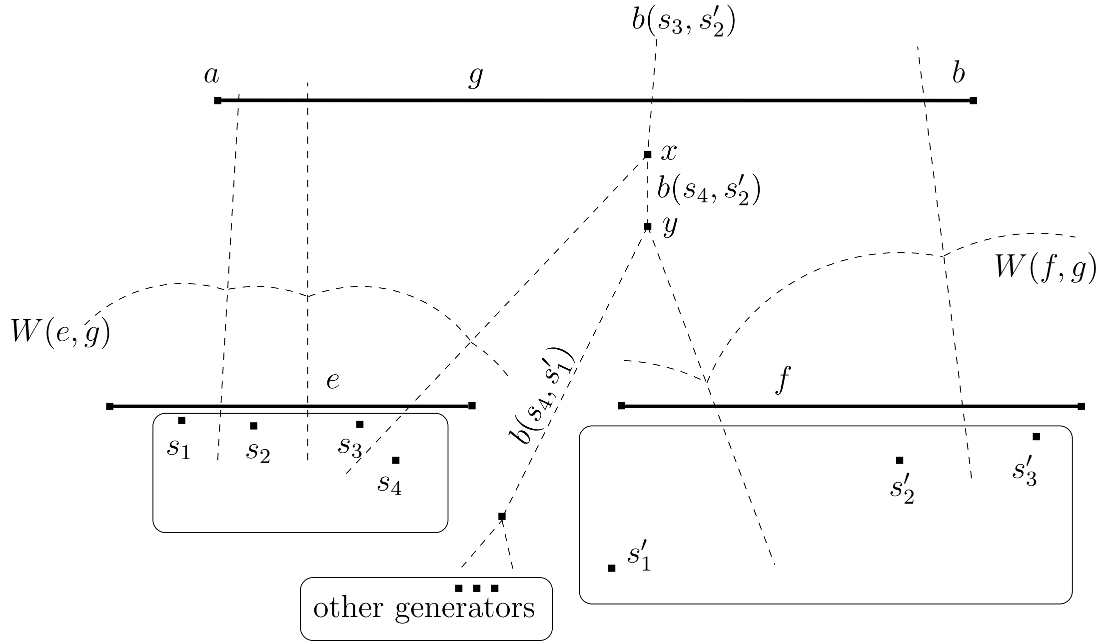

Figure 39: *Examples of bisector events of the second kind, occurring when the waves of $W(e, g)$ collide with those of $W(f, g)$. The wave of the generator s_4 is completely eliminated by s_3 and s'_2 (after s_4 and s'_2 eliminate the wave of s'_1), hence the events x and y occur somewhere between e, f and g, but are not computed during the merge at g. The claims of s_3 and s'_2 on g are shortened by each other, and the new bisector $b(s_3, s'_2)$ is computed by the merging procedure (but its starting event point x is not).*

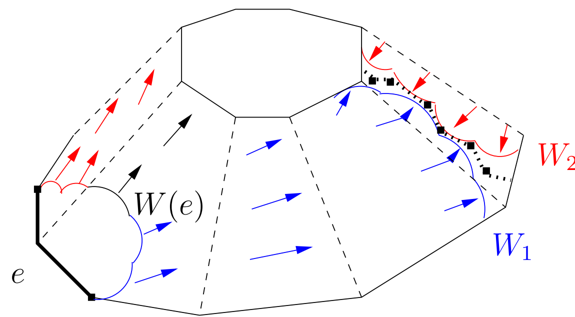

Figure 40: *A wavefront $W(e)$ propagated from e is split inside $R(e)$ when it reaches the inner (top) boundary cycle. Then the two new topologically constrained wavefronts partially collide into each other, creating a sequence of bisectors (dotted lines, bounded by thick points where bisector events of the second kind occur), eliminating a sequence of waves in each wavefront.*

each of the one-sided wavefronts $W(e)$ at e is not a "real wavefront", in the sense of being the locus of waves that traverse a fixed distance from s. Rather, it is a sequence of waves that claim points $q \in e$, where the distances of these points to s are not constant but vary continuously along e. However, a bisector event, which has been considered false when it has been detected beyond e (before e has been ascertained to be fully covered), is detected again, and considered to be true, when the wavefront is propagated further, after processing e. This latter propagation from e, can be considered to start at the time when the first

among such events occurs, which might happen earlier than *covertime*(e); see Figure 41. Further details are given in Section 5, where we also show that the number of all "true" and "false" processed events is only $O(n)$.

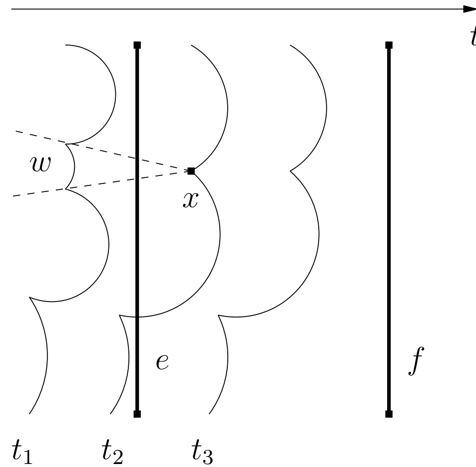

Figure 41: *The bisector event at x occurs at time t_2. It is first detected when the wavefront is propagated toward the transparent edge e, which has not been fully covered yet. Since x is beyond e, the event is currently considered false (and the eliminated wave w is not deleted from the wavefront, so that it shows up on $W(e)$). When e is ascertained (at time $t_3 = covertime(e)$) to be fully covered, the one-sided wavefront $W(e)$ is computed, and then propagated toward the transparent edge f, starting from some time $t < covertime(e)$ (e.g., t_2). Since w is part of $W(e)$, the bisector event at x is detected again, and this time it is considered to be true.*

Remark: Note that a detected "true" event does not necessarily appear as a vertex of SPM(s), since it involves only waves from a single one-sided wavefront, and its location x can actually be claimed by a wave from another wavefront. However, since x belongs to only $O(1)$ well-covering regions, each of which is traversed by only $O(1)$ wavefronts, we can store the separate "trace" of each of them through $R(e)$. Then, when a shortest path query point q is given, we can compare the lengths of the shortest paths to q in each of the "traces" to obtain the true shortest path from s to q. We also show in Section 5 that there are no true events of the second kind between the waves of a single topologically constrained wavefront. From now on, we ignore the bisector events of the second kind, and use the term *bisector event* only for the events of the first kind, unless otherwise specified. See the following discussion and Section 5.4 for further details.

We say that the generators and their waves that are involved in a true bisector event in a well-covering region $R(e)$ are *active in* $R(e)$. As we show in the next section, our algorithm processes only $O(n)$ such events over the entire subdivision. As a result, most waves pass through $R(e)$ unaffected, and leave an "inactive" trace, consisting of a sequence of uninterrupted bisectors, whose first and last elements separate between the active and inactive portions of $R(e)$. That is, we actually subdivide each well-covering region into portions, so that each portion is traversed only by active or only by inactive waves, but not by both. Moreover, we construct $O(1)$ such subdivisions, one for each topologically constrained wavefront that traverses $R(e)$. This will allow us to optimally unite adjacent portions where no (detected) events occur, and will result in a total of only $O(n)$ portions,

with $O(n)$ overall complexity, of (the unfolded) ∂P, which therefore makes it possible to effectively preprocess all regions for point location to answer shortest path queries[13] — see Section 5.4.

The finer details of the propagation algorithm and the merging procedure are described in Section 5, as well as the preprocessing for, and processing of, shortest path queries.

5 Implementation Details

5.1 The data structures

A one-sided wavefront is an ordered list of generators (source images). Our algorithm performs the following three types of operations on these lists (the first two types are similar to those in the data structure of Hershberger and Suri [22]):

1. *List operations*: CONCATENATE, SPLIT, INSERT, and DELETE. Some of these operations are different from their standard counterparts, as described below. Each operation is applied to the list of generators that represents the wavefront at any particular simulation time.
2. *Priority queue operations*: We assign to each generator in the list a priority (as defined below in Section 5.3.1; it is essentially the time at which the generator is eliminated by its two neighbors), and the data structure needs to update priorities and find the minimum priority in the list.
3. *Source unfolding operations*: To compute explicitly each source image s_i in the wavefront at time t, we need to perform the unfolding of the maximal polytope edge sequence of s_i at t. The data structure needs to update the unfoldings as the wavefront advances, and to answer "unfolding queries", that is, to return the queried source image, appropriately unfolded into a specified plane. Another important operation is a SEARCH in the generator list for a claimer of a given query point (without considering other wavefronts or possible visibility constraints); see Figure 42 below. That is, the bisectors between consecutive generators in the list, as long as they do not meet one another, partition a portion of the plane of unfolding into a linearly ordered sequence of regions, and we want to locate the region containing the query point. (We can treat this operation as a variant of standard binary search in a sorted list; see below.)

All these types of operations can be supported by a data structure based on balanced binary search trees, for example, red-black trees, with the generators stored at the leaves [8, 19]. In particular, the "bare" list operations (ignoring the maintenance of priorities and unfolding data) take $O(\log n)$ time each, because the maximum list length is $O(n)$. The

[13]One has to be a little careful here, because well-covering regions can overlap each other. However, since we implement the propagation inside a well-covering region as a sequence of $O(1)$ propagations inside the surface cells that it contains, and each cell is contained in at most $O(1)$ well-covering regions, we can preprocess each cell into at most $O(1)$ separate point location structures.

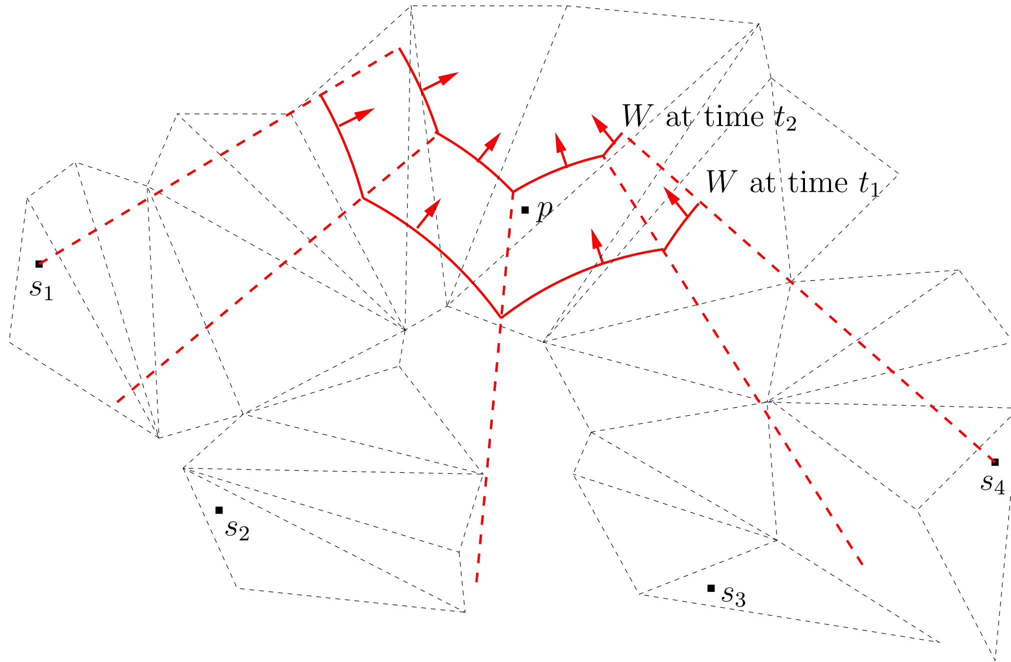

Figure 42: *The wavefront W at simulation times t_1 and t_2 consists of four source images $s_1, \ldots, s_4$, all unfolded to one plane at time t_1 and to another plane at time t_2; for this illustration, both planes are the same — this is the plane of the facet that contains the point p. The bisectors of the wavefront are thick dashed lines (including the two extreme artificial bisectors); in order to determine the generator of W that claims p, the* SEARCH *operation can be applied to the version of W at time t_2, when p is already claimed by s_3.*

priority queue operations are supported by adding a priority field to each node of the binary tree, which records the minimum priority of the leaves in the subtree of that node (and the leaf with that priority). Each priority queue operation takes $O(\log n)$ time (since updating the priority of a leaf entails the updating of only the nodes on its path to the root, and the minimum priority leaf is accessible at the root in $O(1)$ time), while the list operations retain their $O(\log n)$ time bound. The details of maintaining the priority fields during all kinds of tree updates are standard by now (see, e.g., [17] and [22]), and are therefore omitted.

Source unfolding operations. The source unfolding queries are supported by adding an unfolding transformation field $U[v]$ to each node v of the binary tree, in such a way that, for any queried generator s_i, the unfolding of s_i is equal to the product (composition) of the transformations stored at the nodes of the path from the leaf storing s_i to the root. That is, if the nodes on the path are $v_1 = \text{root}, v_2, \ldots, v_k = \text{leaf storing } s_i$, then the unfolding of s_i is given by $U[v_1]U[v_2]\cdots U[v_k]$. We represent each unfolding transformation as a single 4×4 matrix in homogeneous coordinates (see [37]), so composition of any pair of transformations takes $O(1)$ time — see Section 2.1 for details. The unfolding fields have the following property. For each node v, and for any path $v = v_1, v_2, \ldots, v_k$ that leads from v to a leaf, the product $U[v_1]U[v_2]\cdots U[v_k]$ maps the generator stored at v_k to a fixed destination plane that depends only on v.

To perform the SEARCH operation efficiently, we precompute and store at each internal node v of the tree its *bisector image* $b[v]$, defined as follows. Let ($v = v_1, v_2, \ldots, v_k =$ the rightmost leaf of the left subtree of v) denote the sequence of nodes on the path from v to v_k, and let ($v = v'_1, v'_2, \ldots, v'_{k'} =$ the leftmost leaf of the right subtree of v) denote the sequence of nodes on the path from v to $v'_{k'}$. We store at v the bisector $b[v] = b(U[v_1]U[v_2]\cdots U[v_k](s), U[v'_1]U[v'_2]\cdots U[v'_{k'}](s))$, which is the bisector between the source image stored at the rightmost leaf of the left subtree of v and the source image stored at the leftmost leaf of the right subtree of v, unfolded into the destination plane of $U[v_1]U[v_2]\cdots U[v_k]$ (or, *equivalently*, of $U[v'_1]U[v'_2]\cdots U[v'_{k'}]$). Note that, for any path π from v to a leaf in the subtree of v, the *destination plane* $\Lambda(v)$ of the resulting composition of the unfolding transformations stored at the nodes of π, in their order along π, is the same, and depends only on v (and independent of π). As described below, during any operation that modifies the data structure, we always maintain the invariant that $b[v]$ is unfolded onto $\Lambda(v)$.

The procedure SEARCH with a query point q in $\Lambda(\text{root})$ is performed as follows. We determine on which side of $b[\text{root}]$ q lies, in constant time, and proceed to the left or to the right child of the root, accordingly. When we proceed from a node v to its child, we maintain the composition $U^*[v]$ of all unfolding transformations on the path from the root to v (by initializing $U^*[\text{root}] := U[\text{root}]$ and updating $U^*[w] := U^*[u]U[w]$ when processing a child w of a node u on the path). Thus, denoting by b the bisector whose corresponding image $b[v]$ is stored at v, we can determine on which side of b q lies, by computing the image $U^*[v]b[v]$, in $O(1)$ time. Since the height of the tree is only $O(\log n)$, it takes $O(\log n)$ time to SEARCH for the claimer of q.

Initializing the unfolding fields is trivial when the unique singleton wavefront is initialized at $t = 0$ at s. In a typical step of updating some wavefront W, we have a contiguous subsequence W' of W, which we want to advance through a new polytope edge sequence $\mathcal{E}$ (given that all the source images in W are currently unfolded to the plane of the first facet of the corresponding facet sequence of $\mathcal{E}$; see Section 5.3 for further details). We perform two SPLIT operations that split T into three subtrees T^-, T', T^+, where T' stores W', and T^- (resp., T^+) stores the portion of W that precedes (resp., succeeds) W' (either of these two latter subtrees can be empty). Then we take the root r' of T', and replace $U[r']$ by $U_{\mathcal{E}}U[r']$ and $b[r']$ by $U_{\mathcal{E}}b[r']$; see Figure 43 for an illustration. Finally, we concatenate T^-, the new T', and T^+, into a common new tree T.

Remarks:

(i) The collection of the fields $U[v]$ and $b[v]$ in the resulting data structure is actually a dynamic version of the *incidence data structure* of Mount [32], which stores the incidence information between m nonintersecting geodesic paths and n polytope edges. Mount's data structure is a collection of trees, one tree for each polytope edge χ, so that the leaves of the tree of χ correspond to the geodesic paths that intersect χ. By sharing common subtrees, the total space requirement is reduced to $O((n+m)\log(n+m))$. Each node v of this structure stores fields whose roles are similar to our $U[v]$ and $b[v]$, and, by constructing the structure, using special "tying" procedures, so that each tree has $O(\log(n+m))$ height, Mount's data structure [32] supports each SEARCH operation (similar to those supported by our data structure) in $O(\log(n+m))$ time. Our data structure has similar space requirements and

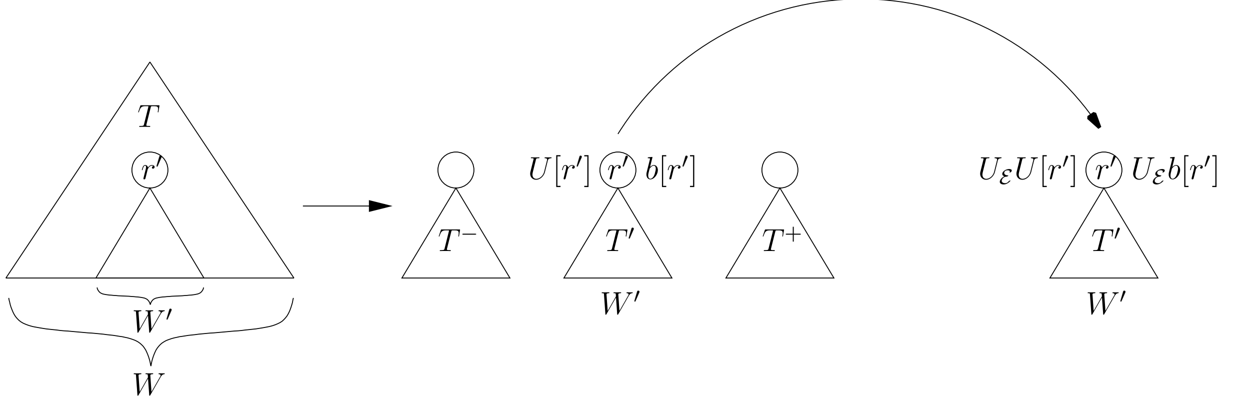

Figure 43: *The tree T is split into three subtrees T^-, T', T^+, where T' stores the sub-wavefront W' of W. Then the unfolding fields stored at the root r' of T' are updated.*

query-time performance; the main novelty is the dynamic nature of the structure and the optimal construction time of $O((n+m)\log(n+m))$. (Mount constructs his data structure in time proportional to the number of intersections between the polytope edges and the geodesic paths, which is $\Theta(nm)$.) In this sense, we combine the benefits of the data structure of Hershberger and Suri [22] with those of Mount [32].

(ii) We could have stored only the unfolding transformations that support the bisector images. Specifically, at each node v we could have stored the two transformations $U[v_1]\cdots U[v_k]$ and $U[v'_1]\cdots U[v'_{k'}]$, where we follow the preceding notation. Using these fields, we can reconstruct the bisector image at a node v in $O(1)$ time, and retrieve the actual unfolding field $U[v]$ (if v is not a leaf; otherwise v stores $U[v]$) also in $O(1)$ time, by computing the product of the unfolding transformation $U[v_1]U[v_2]\cdots U[v_k]$ stored at v with the inverse of that stored at its child v_2 (or, equivalently, by computing the product of the unfolding transformation $U[v'_1]U[v'_2]\cdots U[v'_{k'}]$ with the inverse of that stored at its child v'_2).

(iii) The result of the SEARCH operation is guaranteed to be correct only if the query point q is already covered by the wavefront (that is, the bisectors between consecutive generators in the list do not meet one another closer to s than the location of q). It is the "responsibility" of the algorithm to provide valid query points (in that sense). Recall also that claiming q here is only with respect to the waves in the present wavefront.

Split and concatenate operations. Even though the implementation of these operations are standard by now [19, 41], we discuss them in some detail, to describe how the extra unfolding fields fit into the scheme. (As mentioned above, the maintenance of the priority fields is straightforward, and is not described.) We first describe how to CONCATENATE two trees T_1, T_2 into a common tree T, so that all the leaves of T_1 precede those of T_2. Let r_1, r_2 be the roots of T_1, T_2, respectively; we assume here that $\Lambda(r_1) = \Lambda(r_2)$. For each node u in a red-black tree, denote by $rank(u)$ the number of *black nodes* in a path from u to a leaf, not including u (by definition, all such paths have an equal number of black nodes). Without loss of generality, assume that $rank(r_1) \geq rank(r_2)$.

We follow right pointers from r_1 down the tree T_1 until reaching a black node x with

$rank(x) = rank(r_2)$. Let the rightmost path in T_1 be $r_1 = v_1, v_2, \ldots, v_j = x, v_{j+1}, \ldots, v_k =$ the rightmost leaf of T_1, and let the leftmost path in T_2 be $r_2 = v'_1, v'_2, \ldots, v'_{k'} =$ the leftmost leaf of T_2. We replace x and its subtree by a new red node y, making x the left child and r_2 the right child of y. If r_2 is red, we change its color to black; the black invariant is maintained, since the number of the black nodes in $v_j = x, v_{j+1}, \ldots, v_k$ is $rank(x) + 1$, and it equals (since r_2 is now black) to the number of the black nodes in $r_2 = v'_1, v'_2, \ldots, v'_{k'}$. We store at y the identity transformation $U[y] := I$ and the bisector image

$$b[y] := b(U[v_j]U[v_{j+1}]\cdots U[v_k](s),\ U[v_{j-1}]^{-1}U[v_{j-2}]^{-1}\cdots U[v_1]^{-1}U[v'_1]U[v'_2]\cdots U[v'_{k'}](s)),$$

which is the bisector between the source image stored at the rightmost leaf of T_1 and the source image stored at the leftmost leaf of T_2, unfolded into the destination plane of $U[v_j]U[v_{j+1}]\cdots U[v_k]$ (and of $U[v_{j-1}]^{-1}U[v_{j-2}]^{-1}\cdots U[v_1]^{-1}U[v'_1]U[v'_2]\cdots U[v'_{k'}]$). We update $U[r_2] := U[v_{j-1}]^{-1}U[v_{j-2}]^{-1}\cdots U[v_1]^{-1}U[r_2]$ (here v_{j-1} is the (new) parent of y), and we similarly update $b[r_2] := U[v_{j-1}]^{-1}U[v_{j-2}]^{-1}\cdots U[v_1]^{-1}b[r_2]$. If the parent of y is red, we must recolor and rebalance the tree, from the parent of y up to r_1, as described in [19, 41], to maintain the red invariant. During the rebalancing rotations of the tree, we preserve the correctness of the unfolding information, by performing rotations as illustrated in Figure 44 (only a *single right rotation* is depicted in the figure; a *single left rotation* is performed in a symmetric manner, and a *double rotation* consists of two single rotations; see [19, 41] for more details).

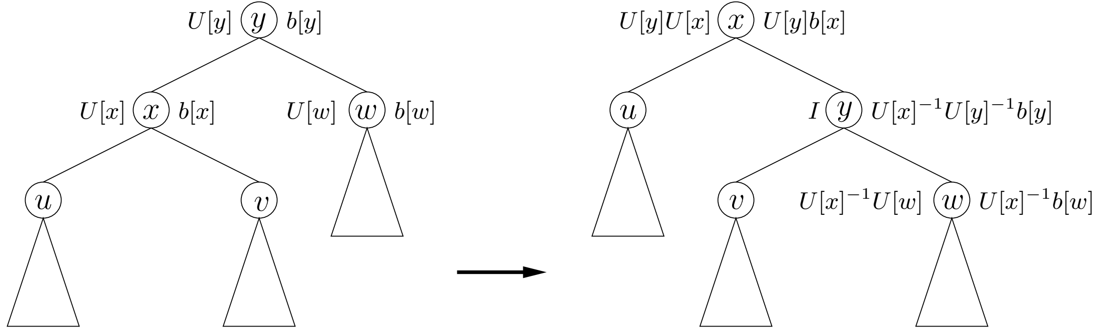

Figure 44: *For each node z, the initial values of its unfolding field $U[z]$ and its bisector image field $b[z]$ appear in the left figure. After rotating the tree to the right, the values are updated as shown in the right figure. Only the fields of the nodes x, y, w change as the result of the rotation. After the rotation, the bisector image field $b[z]$ is unfolded onto a plane that is different from the plane before the rotation, for each $z \in \{x, y, w\}$. For example, $b[x]$ is unfolded onto the destination plane of $U[x]$ before the rotation, and it is unfolded onto the destination plane of $U[y]$ after the rotation; $b[y]$ is unfolded onto the destination plane of $U[y]$ before the rotation, and onto the destination plane of $U[v]$ (which is also the destination plane of $U[x]^{-1}$) after the rotation.*

Notice that the only difference between this and the standard implementations of CONCATENATE (as in [41]) is the updating of the source unfolding information and bisector images (and the priority fields) of the nodes on the path from x to r_1, the rightmost path in the subtree of x, and the leftmost path of T_2, which requires only computations along the $O(\log n)$ nodes on these paths. Hence it still takes $O(\log n)$ time to concatenate two trees, where n is the total number of leaves in both trees.

We next describe how to perform a SPLIT operation on the tree T at a given source image s_i. Denote by T_1 the new tree that has to store the leaves of T up to and including the leaf of s_i, and denote by T_2 the new tree that has to store the leaf of s_i and all succeeding leaves of T. We keep s_i in both portions, since in general the split is caused by some critical event that the wave of s_i reaches (hitting a vertex of P or a transparent endpoint), which causes this wave to be split into two portions, so that the wave of s_i appears in both of the new wavefronts (but with different homotopies when the split is at a vertex of P). We also compute the new artificial bisector (the ray from s_i through the image of the location of the vertex event, as detailed in Section 5.3.1 below), which now becomes an extreme bisector of each of the two new portions of the wavefront.

By following the search path π from the root of T to the leaf storing s_i, we can obtain each of T_1, T_2 as the disjoint union of $O(\log n)$ subtrees of T. The roots of the $O(\log n)$ subtrees that comprise T_1 (resp., T_2) are left (resp., right) children of nodes on π, which themselves lie off π, including the singleton subtree storing s_i in both T_1, T_2.

Since we are going to discard all the nodes of π except the leaf that stores s_i, we first "push away" their unfolding transformations as follows. Let the nodes of π be root $= v_1, \ldots, v_k =$ leaf storing s_i. We traverse π top-down, and compute the products $U^*[v_j] = U[v_1]U[v_2]\cdots U[v_j]$ as we go. When we process v_j, we take the child u_{j+1} which is the sibling of v_{j+1}, update $U[u_{j+1}] := U^*[v_j]U[u_{j+1}]$ and $b[u_{j+1}] := U^*[v_j]b[u_{j+1}]$, and proceed to v_{j+1}. We now cut off the subtrees that hang below π (including the singleton subtree that contains s_i), and discard the nodes of π. We assemble T_1 (resp., T_2) by concatenating the left (resp., right) subtrees (as mentioned above, the singleton storing s_i is concatenated to both T_1, T_2, and becomes the rightmost leaf of T_1, and the leftmost leaf of T_2).

By [41], the time bounds for the CONCATENATE operations that construct each of T_1, T_2 form a telescoping series summing to $O(\log n)$. Updating the nodes that hang below π takes $O(\log n)$ time as well, and so does the manipulation of the unfolding data (and the priorities). Hence our implementation of SPLIT still takes $O(\log n)$ time.

Remark: In fact, in our setup, this implementation replaces the standard INSERT operation, since a new wave is created only when an existing one is split.

Delete operation. We DELETE a generator s_i from a wavefront W either during the merging process or while processing a bisector event; in both cases we are given a pointer to the leaf v of the tree T that stores s_i. We discard v from T; unless the tree becomes empty, we replace the parent x of v by the remaining child y of x. We compute $U_{\text{new}} := U[x]U[y]$ and update $U[y] := U_{\text{new}}$.

If s_i was neither the first nor the last source image in W, then there are two leaves in T that store its neighbors s_{i-1}, s_{i+1}. One of them, say s_{i+1}, must be stored at y; denote by z the leaf that stores s_{i-1}. We traverse the two paths from y and z up to their lowest common ancestor w, composing the two unfoldings U_{i-1}, U_{i+1} stored along the respective paths, as we go. Then we update $b[w]$ (that must have been equal to the appropriate image of $b(s_{i-1}, s_i)$) to $b(U_{i-1}(s), U_{i+1}(s))$.

To maintain the black invariant, we have to rebalance the tree, as described in [19, 41]; the rotations are performed according to the recipe depicted in Figure 44. Again, the only difference between this operation and the standard rotation is the updating of the fields

$U[u], b[u]$ (and the priority fields) of the nodes u on a single path from the deleted leaf to the root, hence the operation still takes $O(\log n)$ time.

Maintaining all versions. We also require our data structure to be *confluently persistent* [14]; that is, we need the ability to maintain, operate on, and modify past versions of any list (wavefront), and we need the ability to *merge* (in the terminology of [14]) existing distinct versions into a new version. Consider, for example, a transparent edge e and two transparent edges f, g in $output(e)$. We propagate $W(e)$ to compute $W(e, f), W(e, g)$; the first propagation has modified $W(e)$, and the second propagation goes back to the old version of $W(e)$ and modifies it in a different manner. Moreover, later, when f, say, is ascertained to be covered, we merge $W(e, f)$ with other wavefronts that have reached f, to compute $W(f)$, and then propagate $W(f)$ further. At some later time g is ascertained to be covered, and we merge $W(e, g)$ with other wavefronts at g into $W(g)$. Thus, not only do we need to retrieve older versions of the wavefront, but we also need to merge them with other versions. All this calls for using a confluently persistent implementation of the structure.

We also use the persistence of the data structure to implement the wavefront propagation through a block tree, as described in Section 5.3.1 below. Specifically, our propagation simulation uses a "trial and error" method; when an "error" is discovered, we restart the simulation from an earlier point in time, using an older version of the wavefront.[14]

Each of the three kinds of operations, CONCATENATE, SPLIT and DELETE, uses $O(1)$ storage for each node of the binary tree that it accesses, so we can make the data structure confluently persistent by path-copying [23]. Each of our operations affects $O(\log n)$ nodes of the tree, including all the ancestors of every affected node. Once we have determined which nodes an operation will affect, and before the operation modifies any node, we copy all the affected nodes, and then modify the copies as needed. This creates a new version of the tree while leaving the old version unchanged; to access the new version we can simply use a pointer to the new root, so traversing it is done exactly as in the ephemeral case. The data structure uses $O(m \log n)$ storage, where m is the total number of operations on the data structure, and keeps the $O(\log n)$ time bound per operation stated above. In summary, we have:

Lemma 5.1. *There exists a data structure that represents a one-sided wavefront and supports all the list operations, priority queue operations, and unfolding operations, as described above, in $O(\log n)$ worst-case time per operation. The size of the data structure is linear in the number of generators; it can be made confluently persistent at the cost of $O(\log n)$ additional storage per operation.*

5.2 Overview of the wavefront propagation stage

Recall from Section 4 that the two main subroutines of the algorithm are wavefront propagation and wavefront merging. In this and the following subsection we describe the implementation details of the first procedure; the merging is discussed in Section 4.2, which,

[14] We do not actually need confluent persistence for that, but, since we already have it, we use it anyway.

together with the data structure details presented in Section 5.1, implies that all the merging procedures can be executed in $O(n \log n)$ time.

Let e be a transparent edge. When the simulation clock reaches time *covertime*(e), we merge all the (at most $O(1)$) contributing wavefronts that have reached e up to this time, separately for each side of e (see Section 4.2), resulting in two one-sided wavefronts $W(e), W'(e)$, which are represented by data structures of the kind just described. We now show how to propagate a given one-sided wavefront $W(e)$ to another edge $g \in$ *output*(e) (that is, $e \in$ *input*(g)), denoting, as above, the resulting propagated wavefronts by $W_{H_1}(e,g), \ldots, W_{H_k}(e,g)$, where $H_1, \ldots, H_k$ are all the relevant homotopy classes of geodesic paths that correspond to block sequences from e to g within $R(g)$ (see Section 3.3); note that a transparent endpoint "splits" a homotopy class, similarly to a vertex of P. In the process, we also determine the time of first contact between each such $W(e,g)$ and the endpoints of g.

The high-level description of the algorithm is a sequence of steps, each of which propagates a wavefront $W(e)$ from one transparent edge e to another $g \in$ *output*(e), within a fixed homotopy class H, to form $W_H(e,g)$. Nevertheless, in the actual implementation, when we start the propagation from e, all the topologically constrained wavefronts $W_H(e,g)$, over all relevant g and H, are treated as *a single wavefront* W. At the beginning of the propagation simulation, W is split into k_1 initial sub-wavefronts, where k_1 is the number of building blocks that e bounds (on the side into which we propagate); during the propagation, these initial wavefronts are further split into a total of k sub-wavefronts, one per homotopy class. The splits occur either at vertices of P, where the current homotopy class is extended to two different classes, or at transparent endpoints, where the splits cut the wavefront into sub-wavefronts, each traversing a different sequence of transparent edges until it eventually reaches a transparent edge of *output*(e).

Let c be the surface cell for which $e \subset \partial c$, and $W(e)$ enters c after reaching e. We describe in the next subsection a procedure for computing (all the relevant topologically constrained wavefronts) $W(e,g)$ for any transparent edge $g \subset \partial c$. Because the edges of *output*(e) belong to a constant number of cells in the vicinity of e, we can use this primitive to compute $W(e,g)$ for all $g \in$ *output*(e), including the edges that do not belong to ∂c, as follows. When we propagate $W(e)$ cell-by-cell inside $R(g)$ from e to g, we effectively split the wavefront into multiple *component wavefronts*, each labeled by the sequence of $O(1)$ transparent edges it traverses from e to g. We propagate a wavefront W from e to g inside a single surface cell, either when W is one of the two one-sided wavefronts merged at e, or when W has reached e on its way to g from some other transparent edge $f \in$ *input*(g) (without being merged with other component wavefronts at e). In what follows, we treat W as in the former case; the latter case is similar.

We also note that the propagation of the initial singleton wavefront (which is the true wavefront, but nonetheless considered as "one-sided" in our procedure) from s to the boundary of the surface cell that contains s, is done exactly as the propagation of a one-sided wavefront $W(e)$ from a transparent edge e to the boundary of a cell that contains e, replacing the Riemann structure $\mathcal{T}(e)$ by the corresponding structure $\mathcal{T}(s)$. This requires simple and obvious modifications which we will not spell out.

5.3 Wavefront propagation in a single cell

So far we have considered a wavefront as a static structure, namely, as a sequence of generators that reach a transparent edge. We now describe a "kinetic" form of the wavefront, in which we track changes in the combinatorial structure of the wavefront $W(e)$ as it sweeps from its origin transparent edge e across a single cell c. Our simulation detects and processes any bisector event in which a wave of $W(e)$ is eliminated by its two neighboring waves inside c; actually, the propagation may also detect some events that occur in $O(1)$ nearby cells, as described in detail below. Events are detected and processed in order of increasing distance from s, that is, in simulation time order. However, *the simulation clock* t *is not updated during the propagation inside* c; that is, the propagation from an edge e to all the edges in $output(e)$ is done without "external interruptions" of propagating from other fully covered transparent edges that need processing. The effect of the propagated wavefront $W(e, g)$, for $g \in output(e)$, on the simulation clock is in its updating of the values $covertime(g)$; the actual updating of t occurs only when we select a new transparent edge e' with minimum $covertime(e')$ for processing — see Section 4.1.

We propagate the wavefront separately in each of the $O(1)$ block trees of the Riemann structure $\mathcal{T}(e)$ (see Section 3.2). Let $W(e)$ be the one-sided wavefront that reaches e from outside c; it is represented as an ordered list of source images, each claiming[15] some (contiguous and *nonempty*) portion of e. To prepare $W(e)$ for propagation in c, we first SPLIT $W(e)$ into $O(1)$ sub-wavefronts, according to the subdivision of e by building blocks of c. A sub-wavefront that claims the segment of e that bounds a building block B of c is going to be propagated in the block tree $T_B(e) \in \mathcal{T}(e)$.

By propagating $W(e)$ from e in all the trees of $\mathcal{T}(e)$ within c, we compute $O(1)$ new component wavefronts that reach other transparent edges of ∂c. If e is the initial edge in this propagation step, then, by Corollary 3.18, these component wavefronts collectively encode all the shortest paths from s to points p of c, which enter c through e and do not leave c before reaching p. In general, this property holds for all the cells c' in $R(e)$, as follows easily from the construction. Hence, these component wavefronts, collected over all propagation steps that traverse c, contain all the needed information to construct (an implicit representation of) SPM(s) within c.

5.3.1 Wavefront propagation in a single block tree

Let $T_B(e)$ be a block tree in $\mathcal{T}(e)$, and denote by e_B the sub-edge $\partial B \cap e$. Denote by $W(e_B)$ the sub-list of generators of $W(e)$ that claim points on e_B (recall that $W(e)$ claims a single connected portion of e, which may or may not contain the endpoints of e, or of e_B). Let $W = W(t)$ denote the kinetic wavefront within the blocks of $T_B(e)$ at any time t during the simulation; initially, $W = W(e_B)$. Note that even though we need to start the propagation from e at simulation time $covertime(e)$, the actual starting time may be strictly smaller, since there may have been bisector events beyond e that have occurred before time $covertime(e)$,

[15]We say here that a generator s' in a wavefront W *claims* a point $p \in \partial P$ if the wave of W represented by s' encodes a geodesic path from s' to p that reaches p before any other path encoded in W, even if the true claimer of p is not in W.

and these events need now to be processed; up to now, they have been detected by the algorithm but not processed yet. The starting time t_0 is the time when the earliest among these events takes place (if there are no such events, $t_0 = covertime(e)$).

Denote by $\mathcal{E}_B$ an edge sequence associated with B (any one of the two oppositely ordered such sequences, for blocks of type II, III), and by $\mathcal{F}_B$ its corresponding facet sequence. We can then write $W = (s_1, s_2, \ldots, s_k)$, so that, for each i, we have $s_i = U_{\mathcal{E}_i}(s)$, where $\mathcal{E}_i$ is defined as follows. Denote by $\tilde{\mathcal{E}}_i$ the maximal polytope edge sequence traversed by the wave of s_i from s_i to the points that it claims on e; $\tilde{\mathcal{E}}_i$ must overlap either with a portion of $\mathcal{E}_B$ or with a portion of the reverse sequence $\mathcal{E}_B^{rev}$. In the former case we extend $\tilde{\mathcal{E}}_i$ by the appropriate suffix of $\mathcal{E}_B$ (which takes us to f in Figure 45). In the latter case we truncate $\tilde{\mathcal{E}}_i$ at the first polytope edge of $\mathcal{E}_B^{rev}$ that it meets, and then extend it by the appropriate suffix of $\mathcal{E}_B$. However, the algorithm does not compute these sequences explicitly (and does not perform the "extend" or "truncate" operations). In fact, the algorithm never manipulates edge sequences explicitly; it only stores and composes their unfolding transformations. The unfolding transformations $U_{\mathcal{E}_i}$ are thus only implicitly maintained, as described in Section 5.1. In this way, the algorithm efficiently unfolds all source images of W onto a common plane (of the last facet of $\mathcal{F}_B$), which we denote by $\Lambda(W)$; we do not alter $\Lambda(W)$ until the propagation of W in $T_B(e)$ is stopped (and then $\Lambda(W)$ is updated, as described below). When we propagate the initial singleton wavefront directly from s in $T_B(s)$, we initialize $W := (s)$, so that the maximal polytope edge sequence of s is empty, and the corresponding unfolding transformation is the identity transformation I. This setting is appropriate since s is assumed to be a vertex of P, and therefore all the polytope edges in $\mathcal{E}_B$ emerge from s, so it lies on all the facets of $\mathcal{F}_B$, and, particularly, on the last facet of $\mathcal{F}_B$.

The wavefront W is propagated into blocks of $T_B(e)$, starting from B and passing from one block to another through the contact intervals that connect them.

Unless otherwise specified, in what follows we treat each node of $T_B(e)$ as a *distinct* building block, even though this block might appear more than once in the tree. Each contact interval between a parent and its child in $T_B(e)$ is also treated as being distinct from any other occurrence of the same contact interval that might show up elsewhere in $T_B(e)$. Transparent edges and block vertices are treated similarly.

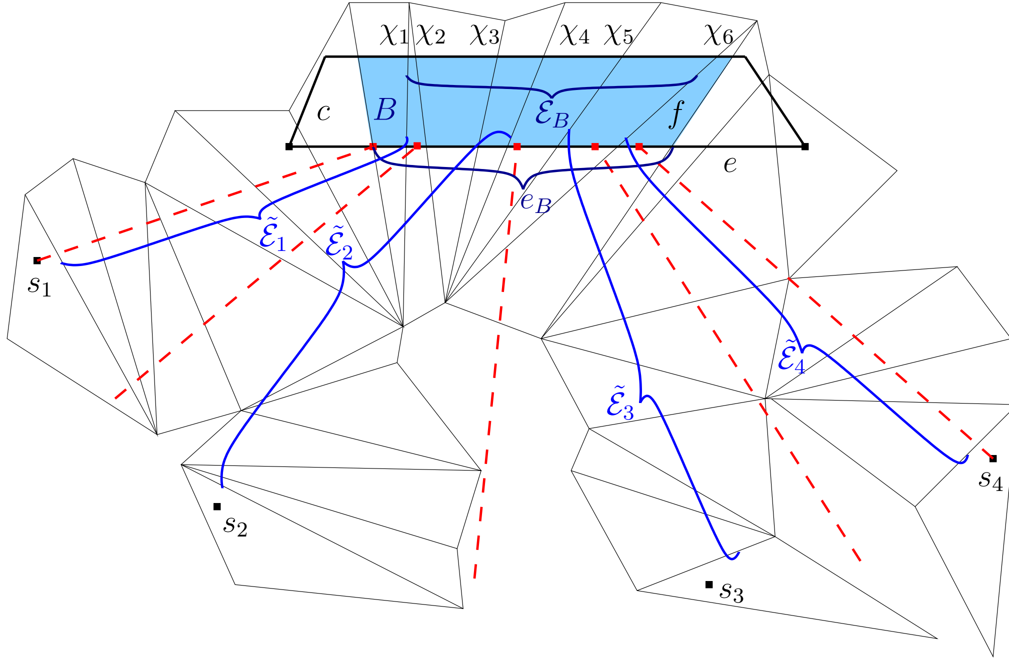

Figure 45: *The block B is shaded; the edge sequence associated with B is $\mathcal{E}_B = (\chi_1, \ldots, \chi_6)$. The bisectors of the wavefront $W(e_B)$ are thick dashed lines (including the two extreme artificial bisectors); $W(e_B)$ consists of four source images $s_1, \ldots, s_4$, all unfolded to the plane of the facet f before the simulation of the propagation into $T_B(e)$ starts (that is, the last facet of the facet sequence corresponding to each $\mathcal{E}_i$ is f). Specifically, $\mathcal{E}_1 = \tilde{\mathcal{E}}_1 || (\chi_2, \ldots, \chi_6)$, $\mathcal{E}_2 = \tilde{\mathcal{E}}_2 || (\chi_5, \chi_6)$, $\mathcal{E}_3 = \tilde{\mathcal{E}}_3 \setminus (\chi_6, \chi_5)$ and $\mathcal{E}_4 = \tilde{\mathcal{E}}_4 \setminus (\chi_6)$.*

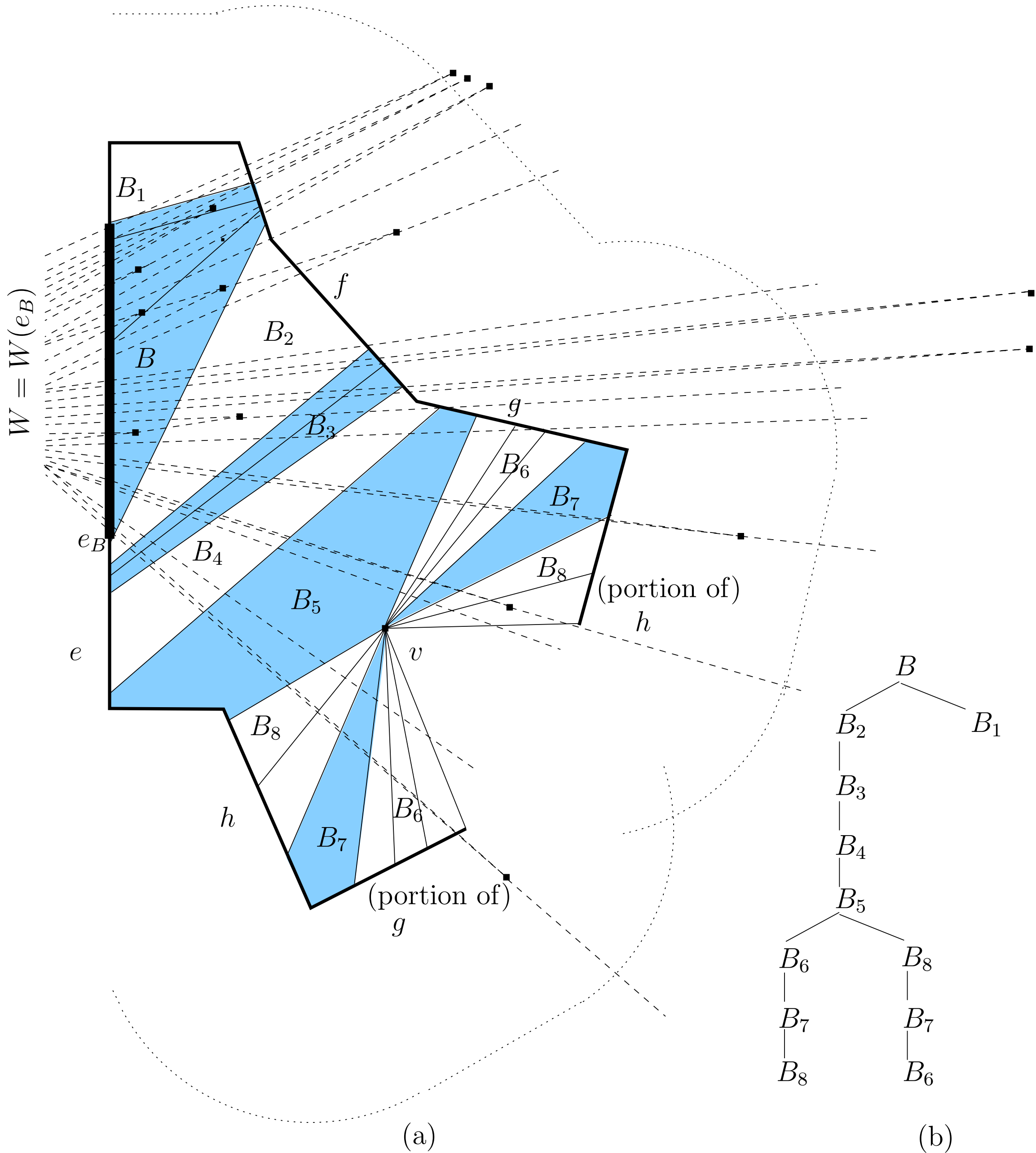

Figure 46: *(a) Bisector events (the thick square points), some of which are processed during the propagation of the wavefront W from the transparent edge portion e_B (the thickest segment in this figure) through the building blocks (their shadings alternate) of the block tree $T_B(e)$ (shown in (b)). The unfolded transparent edges are drawn as thick solid lines, while the unfolded contact intervals are thin solid lines. The bisectors of the generators of W, as it sweeps through the unfolded blocks, are shown dashed. The union of all the blocks in $T_B(e)$ is bounded by e_B and the boundary chain $\mathcal{C}$ (which is non-overlapping in this example). The dotted lines indicate the distance from the transparent edges in $\mathcal{C}$ within which we still process bisector events of W. For each transparent edge f of $\mathcal{C}$, we can stop propagating the wavefront portion $W(e_B, f)$ that has reached f after it crosses the dotted line (which lies at distance $2|f|$ from f), since f must have already been fully swept at that time by the waves of $W(e_B, f)$.*

The *boundary chain* $\mathcal{C}$ of $T_B(e)$ is recursively defined as follows. Initially, we put in $\mathcal{C}$ all the boundary edges of ∂B, other than e_B. We then proceed top-down through $T_B(e)$. For each node B' of $T_B(e)$ and for each child B'' of B', we remove from the current $\mathcal{C}$ the contact interval connecting B' and B'', and replace it by the remaining boundary portion of B''. This results in a connected (unfolded) polygonal boundary chain that shares endpoints with $B \cap e$. (We remind the reader that the unfolding process that produces $\mathcal{C}$ generates a Riemann surface that may overlap itself; such overlaps however are overcome (and essentially ignored) by the local propagation mechanism, as described below.) Since $T_B(e)$ has $O(1)$ nodes, and each block has $O(1)$ boundary elements, $\mathcal{C}$ contains only $O(1)$ elements. See Figure 46 for an illustration of an unfolded $T_B(e)$ and its (unfolded) boundary chain.

When W is propagated towards $\mathcal{C}$, the most important property is that each transparent edge or contact interval of $\mathcal{C}$ can be reached only by a *single topologically constrained sub-wavefront* of W, since, if W splits on its way, the new sub-wavefronts reach different elements of $\mathcal{C}$. Note that the property does not hold for ∂c, since, when c contains holes and/or a vertex of P, there is more than one way to reach a transparent edge $f \in \partial c$ — in such cases f appears more than once in $\mathcal{C}$, each time as a distinct element (this is illustrated in Figure 46). In the rest of this section, whenever a resulting wavefront $W(e, f)$ is mentioned for some $f \in \mathcal{C}$, we interpret $W(e, f)$ as $W_H(e, f)$ for the unique homotopy class H that constrains W on its way from e to this specific incarnation of f along $\mathcal{C}$.

We denote by *range*(W) the portion of $\mathcal{C}$ that can potentially be reached by W, initialized as *range*$(W) := \mathcal{C}$. As W is propagated (and split), *range*(W) is updated (that is, split and/or truncated) accordingly, as described below.

Critical events and simulation restarts. We simulate the continuous propagation of W by updating it at the (discrete) critical events that change its topology during its propagation in $T_B(e)$. There are two types of these events — bisector events (of the first kind), when a wave of W is eliminated by its two neighbors, and vertex events, when W reaches a vertex of $\mathcal{C}$ (either transparent or a real vertex of P) and has to be split. Before we describe in detail the exact processing of all the cases that may arise, we provide here a high-level description of these cases, and the intuition behind the (somewhat unorthodox implementation of the) low-level procedures.

The purpose of the propagation of W in $T_B(e)$ is the computation of the wavefronts $W(e_B, f)$, for each transparent edge f in $\mathcal{C}$ that W reaches. To do so, we have to correctly update W at those critical events that are *true events with respect to the propagation of* W *in* $T_B(e)$, i.e., the events that take place in $T_B(e)$ that would have been vertices of SPM(s) if there were no other wavefronts except W to propagate through the blocks of $T_B(e)$. For the sake of brevity, in the rest of this section we call these events simply *true events*. Unfortunately, it is difficult to determine *in advance* the exact set of true events (mainly because of vertex events — see below). Instead, we determine on the fly a larger set of *candidates* for critical events, which is guaranteed to contain all the true events, but which might also contain events that are *false with respect to the propagation of* W *in* $T_B(e)$; in the rest of this section we refer to events of the latter kind as *false events*. The candidates that turn out to be false events either are bisector events that involve at least one generator s' of W so that the path from s' to the event location intersects $\mathcal{C}$, or are computed based

on incomplete information of earlier true events (at least one of which has not been detected and processed in time). (Note that this classification differs from the ultimate classification of events as "true" or "false" according to whether an event is or is not a vertex of SPM(s). Instead, the algorithm prunes away events that it can ascertain not to take place in $T_B(e)$, and regards the remaining ones as (tentatively) true.)

Let x be such a *candidate bisector event* that takes place at simulation time t_x. If all the true events of W that *have taken place before* t_x were *processed before* t_x, then x can be *foreseen* at the last critical event at which one of the bisectors involved in x was updated before time t_x, using the *priorities* assigned to the source images in W. The priority of a source image s' is the distance from s' to the point at which the two (unfolded) bisectors of s' (one of which is artificial, if s' is extreme in W) intersect beyond e_B, either in B or beyond it. (In the latter case we cannot right away locate the intersection point, because it may depend on polytope edge sequences that "continue the unfolding", which are not immediately available; we explain below how we overcome this problem by explicitly "tracing" the involved paths to the location of x beyond B through $T_B(e)$.) The priority is $+\infty$ if the bisectors do not intersect beyond e_B. (Initially, when W contains the single wave from s, the priority of s is defined to be $+\infty$.) Whenever a bisector of a source image s' is updated (as detailed below), the priority of s' is updated accordingly.

A *candidate vertex event* cannot be foreseen so easily, since we do not know which source image of W claims a vertex v (because of the critical events that might change W before it reaches v), until v is actually reached by W. Even when v is reached by W, we do not have in the data structure a "warning" that this vertex event is about to take place. Instead, we detect the vertex event that occurs at v only later and indirectly, either when processing some later candidate event (which is false as it was computed without taking into account the event at v — see Figure 47(a,b)), or when the propagation of W in $T_B(e)$ is stopped at a later simulation time, when a segment f of $\mathcal{C}$ incident to v is ascertained to be fully covered (in which case we want to split out from W the sub-wavefront W' that claims f, since W' must not be propagated further, as described below), as illustrated in Figure 47(c). Both cases are detailed further in this section.

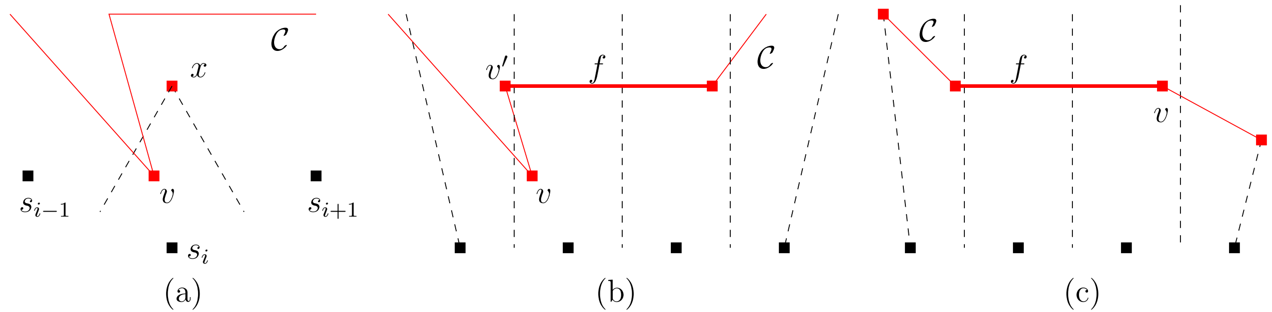

Figure 47: *A vertex event at a vertex v of $\mathcal{C}$, which has been reached by the wavefront W at some earlier time t_v, can be detected: (a) while processing a false bisector event x at the later time priority(s_i); (b) while processing a vertex event at an endpoint v' of a segment f of $\mathcal{C}$, at some later time when f is ascertained to be covered by W; (c) when the segment f of $\mathcal{C}$, incident to v, is ascertained to be covered by W.*

Here and later in this section, we denote by *claimer*(p), for a point p in a block of $T_B(e)$,

the generator visible from p (in the corresponding unfolded block sequence of $T_B(e)$) which is nearest to p among all such generators of $W(e_B)$.

When we detect a vertex event at some vertex v which is reached by W at time t_v, so that at least one candidate critical event of W, which takes place later than t_v, has already been processed, *all the versions of the (persistent) data structure that encode W after time t_v become invalid, since they do not reflect the update that occurs at t_v.* To correct this situation, we discard all the invalid versions of W, and *restart the simulation of the propagation of the last valid version of W from time t_v.* This time, however, we split W at v (at simulation time t_v) into two new sub-wavefronts, making the ray from *claimer*(v) to v the new (artificial) extreme bisector of both, as detailed below. Note that this step does not guarantee that the current event at v is a true event, since there might still exist undetected earlier vertex events, which, when eventually detected later, will cause the simulation to be restarted again, making the current event at v invalid (and we will have to wait until the wavefront reaches v again). In spite of all this overhead, we will argue below that these restarts do not affect the asymptotic time complexity of the propagation of W.

In other words, the processing of the critical events, described below, is *valid* at a given simulation time t (that is, the wavefront that is maintained by the algorithm at time t contains all the necessary shortest paths and does not include invalid paths that violate visibility constraints or are longer than alternative paths within W) only under the assumption that all the vertex events that had taken place before t have already been (correctly) detected and processed — otherwise, at each such future detection of a "past" vertex event that has occurred at some time $t' < t$, the simulation process will be restarted from time t'.

Path tracing. Let x be the (unfolded) location of a candidate critical event. To determine whether the path to x from its claimer (or the paths from its claimers, if x is a bisector event) does or does not intersect[16] $\mathcal{C}$, and, in the former case, to also determine the first intersection point (along the path) with $\mathcal{C}$, we use the following *path tracing procedure*. It receives as input a source image s' and the image of x unfolded onto $\Lambda(W)$, and traces $\pi(s', x)$ either up to x, or until $\pi(s', x)$ intersects $\mathcal{C}$ — whichever occurs first.

The tracing is done as follows. We first compute the unfolded image of ∂B (onto $\Lambda(W)$), and consider the following cases.

If $\pi(s', x)$ does not intersect $\partial B \setminus e_B$ before reaching x, then x lies in B, and we are done. If $\pi(s', x)$ intersects a transparent edge of $\partial B \setminus e_B$ before reaching x, then we are also done, since we have reached $\mathcal{C}$.

Suppose next that $\pi(s', x)$ intersects some contact interval I of ∂B before intersecting any transparent edge of $\partial B \setminus e_B$ (and before reaching x). If B does not have a child in $T_B(e)$ that is connected to B through I, then I belongs to $\mathcal{C}$, and we are done: The candidate event at x is false; that is, there must be some other wave, reaching the region on the other

[16]Here and in the rest of this section, whenever we say that a path π *intersects* $\mathcal{C}$ at a point p along its way to a critical event x, we include the case where π merely *touches* $\mathcal{C}$ at p, without crossing, if p is not an endpoint of $\mathcal{C}$; the reason is that in this case p must be a vertex of $\mathcal{C}$, and therefore we detect a vertex event at p that takes place before x. (However, these cases can be ignored by assuming general position.)

side of I, that reaches x earlier and claims it, by Corollary 3.18.

Otherwise, we pass to the unique child B' of B in $T_B(e)$ that is connected to B through I, and repeat this step. In more detail, denote by $\mathcal{B}$ the block sequence (B, B'), and denote by $\mathcal{E}$ the edge sequence associated with $\mathcal{B}$ ($\mathcal{E}$ is unique — see Section 3.2). We find the unfolded images $U_{\mathcal{E}}(\pi(s', x))$ and $U_{\mathcal{E}}(B')$. (Note that we do not yet update the unfolding transformation of s' in the source unfolding data structure; all such updates will be done at the end of the propagation of W in $T_B(e)$ — see below.) Then we can determine whether x lies in B', or whether $\pi(s', x)$ intersects $\partial B' \setminus I$ before reaching x, similarly to the procedure described above for $\partial B \setminus e_B$, and recursively proceed to further building blocks, as above.

At each step we proceed in $T_B(e)$ from a node to its child; since the depth of $T_B(e)$ is $O(1)$, we are done after $O(1)$ steps. Since at each step we compute $O(1)$ unfoldings of paths and transparent edges, and each unfolding operation takes $O(\log n)$ time to perform, using the data structures described in Sections 2.4 and 5.1, the whole tracing procedure takes $O(\log n)$ time.

Corollary 5.2. *Tracing the path $\pi(s', p)$ from a generator $s' \in W$ to a point p without intersecting $\mathcal{C}$, correctly determines the distance $d(s', p)$.*

Proof: Since $\mathcal{C}$ is not intersected, $\pi(s', p)$ is a valid geodesic path that traverses a valid polytope edge sequence corresponding to s'; distances to p from other source images (which may be closer to p than s' is) do not change this fact. $\square$

Remark: Although the unfolding of the block sequence $\mathcal{B}$ in a root-to-leaf path of $T_B(e)$ might overlap itself, it does not affect the above tracing procedure, which traverses $\mathcal{B}$ block-by-block, each time computing the intersection of a (straight-line) unfolded image with the unfolded boundary of the next block of $\mathcal{B}$.

Since any shortest path from s that enters B through e and does not stop inside c must leave c through $\mathcal{C}$, after crossing $O(1)$ building blocks, we can also use the above procedure to trace any such path of W until it intersects $\mathcal{C}$, without specifying any terminal point on the path, as long as the starting direction of the path in the plane is well defined.

In the rest of this section, whenever we say that a *path* π from a generator $s' \in W$ intersects $\mathcal{C}$, we actually mean that only the portion of π from s' to the first intersection point $x = \pi \cap \mathcal{C}$ is a valid geodesic path; the portion of π beyond x is merely a straight segment along the direction of π on $\Lambda(W)$. Still, for the sake of simplicity, we call π (including possibly a portion beyond x) a *path* (from s' to the terminal point of π).

The following technical lemma is needed later for the correctness analysis of the simulation algorithm — in particular, for the analysis of critical event processing. See Figure 48.

Lemma 5.3. *Let s_i, s_j be a pair of generators in W, and let p_i, p_j be a pair of (possibly coinciding) points in the unfolded blocks of $T_B(e)$, so that $\pi(s_i, p_i)$ and $\pi(s_j, p_j)$ do not intersect each other (except possibly at their terminal point, if $p_i = p_j$), and if $p_i \neq p_j$ then $f = \overline{p_i p_j}$ is a straight segment of $\mathcal{C}$. Denote by z_i (resp., z_j) the intersection point $\pi(s_i, p_i) \cap e_B$ (resp., $\pi(s_j, p_j) \cap e_B$), and denote by τ the unfolded convex quadrilateral (or triangle) $z_i p_i p_j z_j$. Let B' be the last building block of the maximal common prefix block sequence along which both $\pi(s_i, p_i)$ and $\pi(s_j, p_j)$ are traced (before possibly diverging into different blocks).*

If only one of the two paths leaves B', or if $\pi(s_i, p_i)$ and $\pi(s_j, p_j)$ leave B' through different contact intervals of $\partial B'$, then the region $B' \cap \tau$ contains at least one vertex of $\mathcal{C}$ that is visible, within the unfolded blocks of $T_B(e)$, from every point of $\overline{z_1 z_2} \subseteq e_B$.

Proof: Assume for simplicity that $B' \neq B$. The paths $\pi(s_i, p_i), \pi(s_j, p_j)$ must enter B' through a common contact interval I of $\partial B'$. Consider first the case where $\pi(s_i, p_i), \pi(s_j, p_j)$ leave B' through two respective different contact intervals I_i, I_j of $\partial B'$, and denote their first points of intersection with $\partial B'$ by x_i and x_j, respectively — see Figure 48(a). Denote by $\mathcal{X}$ the portion of $\partial B'$ between x_i and x_j that does not contain I; $\mathcal{X}$ must contain at least one vertex of $\partial B'$. By definition, each vertex of a building block is a vertex of $\mathcal{C}$; note that the extreme vertices of $\mathcal{X}$ are x_i and x_j, which may or may not be vertices of $\mathcal{C}$. Since the unfolded image of $\mathcal{X}$ is a simple polygonal line that connects $\pi(s_i, x_i)$ and $\pi(s_j, x_j)$, and intersects neither $\pi(s_i, x_i)$ nor $\pi(s_j, x_j)$, it is easily checked that we can sweep τ by a line parallel to e_B, until we encounter a vertex v of $\mathcal{X}$ within τ, which is also a vertex of $\mathcal{C}$: Either x_i or x_j is such a vertex, or else τ must contain an endpoint of either I_i or I_j. Therefore v is visible from each point of $\overline{z_1 z_2}$.

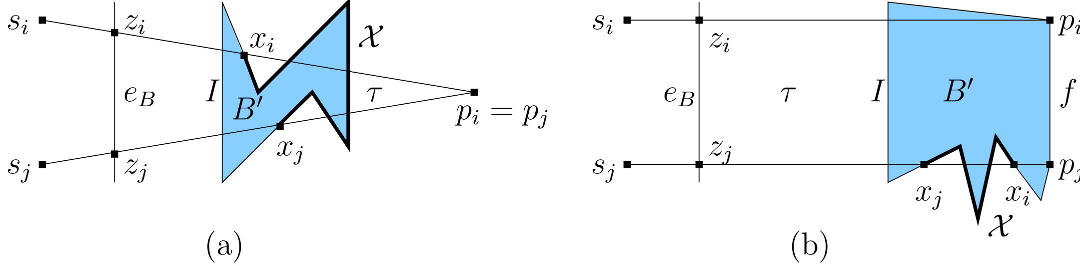

Figure 48: *(a) $\pi(s_i, p_i), \pi(s_j, p_j)$ leave B' through two different contact intervals of $\partial B'$. In this example $p_i = p_j$, and τ is the triangle $z_i p_i z_j$. (b) $\pi(s_i, p_i)$ reaches $p_i \in B'$ and $\pi(s_j, p_j)$ leaves B' at the point x_j. In this example $p_i \neq p_j$, and τ is the quadrilateral $z_i p_i p_j z_j$. The portion $\mathcal{X}$ of $\partial B'$ is highlighted in both cases.*

Consider next the case where only one of $\pi(s_i, p_i), \pi(s_j, p_j)$ leaves B', and assume, without loss of generality, that $\pi(s_i, p_i)$ reaches $p_i \in B'$ and $\pi(s_j, p_j)$ leaves B' at the point x_j before reaching p_j — see Figure 48(b). Denote by $\pi(p_j, s_j)$ the path $\pi(s_j, p_j)$ directed from p_j to s_j, and denote by π' the concatenation $\pi(s_i, p_i)||\overline{p_i p_j}||\pi(p_j, s_j)$. The path $\pi(s_i, p_i)$ does not leave B', and, by assumption, the segment $\overline{p_i p_j}$ is either an empty segment or a segment of $\partial B'$, and therefore the only portion of π' that leaves B' is $\pi(p_j, s_j)$. Denote by x_i the first point along $\pi(p_j, s_j)$ (beyond p_j itself) that lies on $\partial B'$; if $\pi(p_j, s_j)$ leaves B' immediately, we do take $x_i = p_j$. Since (the unfolded) $\pi(p_j, s_j)$ is a straight segment, and since, for each segment f' of $\partial B'$, B' lies locally only on one side of f', it follows that x_i and x_j lie on different segments of $\partial B'$. Define $\mathcal{X}$ as above; here it connects the prefixes of π' and $\pi(s_j, p_j)$, up to x_i and x_j, respectively, and the proof continues as in the previous case. $\Box$

Stopping times and their maintenance. The simulation of the propagation of W in the blocks of $T_B(e)$ processes candidate bisector events in order of increasing priority, up to some

time $t_{\text{stop}}(W)$, which is initialized to $+\infty$, and is updated during the propagation.[17] When the time $t_{\text{stop}}(W)$ is reached, the following holds: Either $t_{\text{stop}}(W) = +\infty$ (see Figure 49(a)), all the known candidate critical events of W in the blocks of $T_B(e)$ have been processed, and all the waves of W that were not eliminated at these events have reached $\mathcal{C}$; or $t_{\text{stop}}(W) < +\infty$ (see Figure 49(b)), and there exists some sub-wavefront $W' \subseteq W$, which claims some segment (a transparent edge or a contact interval) f of $range(W)$ (that is, f is ascertained to have been covered by W' not later than at time $t_{\text{stop}}(W)$), such that all the currently known candidate events of W' have been processed before time $t_{\text{stop}}(W)$. In the former case we split W into sub-wavefronts $W(e, f)$ for each segment $f \in range(W)$; in the latter case, we extract from W (by splitting it) the sub-wavefront $W(e, f) = W'$ that has covered f. When we split W into a pair of sub-wavefronts W_1, W_2, the time $t_{\text{stop}}(W_1)$ (resp., $t_{\text{stop}}(W_2)$) replaces $t_{\text{stop}}(W)$ in the subsequent propagation of W_1 (resp., W_2), following the same rule, while $t_{\text{stop}}(W)$ plays no further role in the propagation process.

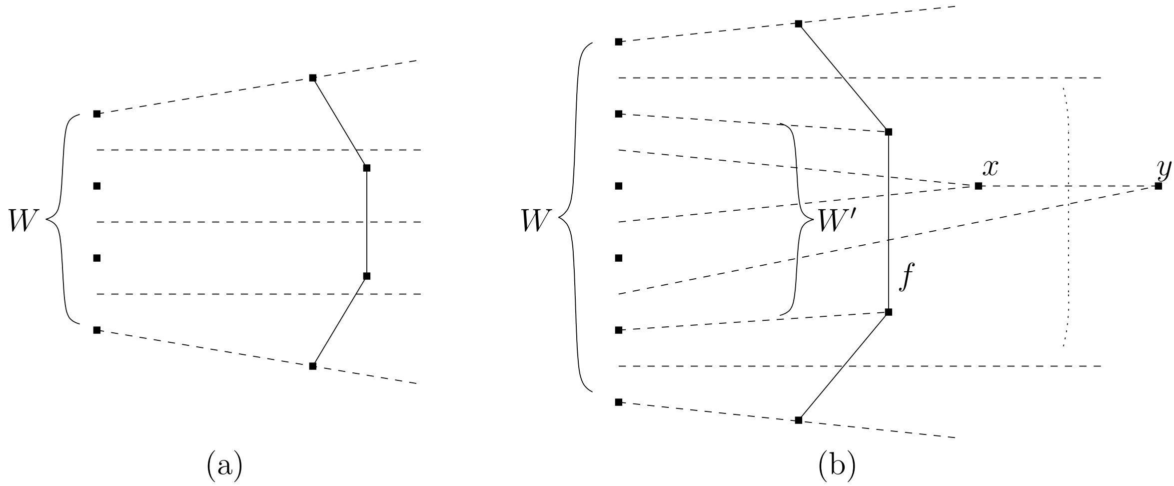

Figure 49: *(a) The stopping time $t_{\text{stop}}(W) = +\infty$. (b) The stopping time $t_{\text{stop}}(W') = t_{\text{stop}}(f) < +\infty$; the dotted line indicates the stopping time (or distance) at which we stop processing bisector events: the event at x has been processed before $t_{\text{stop}}(W')$, while the event at y has been detected but not processed.*

For each segment f in $\mathcal{C}$, we maintain an individual time $t_{\text{stop}}(f)$, which is a conservative upper estimate of the time when f is completely covered by W during the propagation in $T_B(e)$. Initially, we set $t_{\text{stop}}(f) := +\infty$ for each such f. As detailed below, we update $t_{\text{stop}}(f)$ whenever we trace a path from a generator in W that reaches f (without reaching $\mathcal{C}$ beforehand); by Corollary 5.2, these updates are always valid (i.e., do not depend on simulation restarts).

The time $t_{\text{stop}}(W)$ is the minimum of all such times $t_{\text{stop}}(f)$, where f is a segment of $range(W)$. Whenever $t_{\text{stop}}(f)$ is updated for such an f, we also update $t_{\text{stop}}(W)$ accordingly. When the simulation clock reaches $t_{\text{stop}}(W)$, either some f of $range(W)$ is completely covered by the wavefront W, so that $t_{\text{stop}}(f) = t_{\text{stop}}(W)$, or the priority of the next event of W in the priority queue is $+\infty$, in which case $t_{\text{stop}}(W) = +\infty$.

[17]The present description also applies to appropriate sub-wavefronts that have already been split from W — see below.

As shown below, $range(W)$ is maintained correctly, independently of simulation restarts; therefore, when $range(W)$ contains only one segment, no further vertex events may cause a restart of the simulation of the propagation of W (since a simulation restart of a wavefront that is separated from W does not affect W, and the vertex events at the endpoints of f have already been processed, since W and $range(W)$ have already been split at them).

Before going into the details of the stopping time maintenance procedures, we explain the intuition behind them.

First, note that there is a gap of at most $|f|$ time between the time t_f when the segment f of $\mathcal{C}$ is *first reached* by W and the time when f is *completely covered* by W. In particular, it is possible that both endpoints of f are reached by W before f is completely covered by W — see Figure 50(a) for an illustration. It is also possible, because of visibility constraints, that W reaches only a portion of f in our propagation algorithm (and then there must be other topologically constrained wavefronts that reach the portions of f that are not reached by W). Still, if f is reached by W at some time t_f, we say that f is *covered by* W at time $t_f + |f|$, as if we were propagating also the non-geodesic paths that progress along f from the first point of contact between W and f. See Figure 50(b).

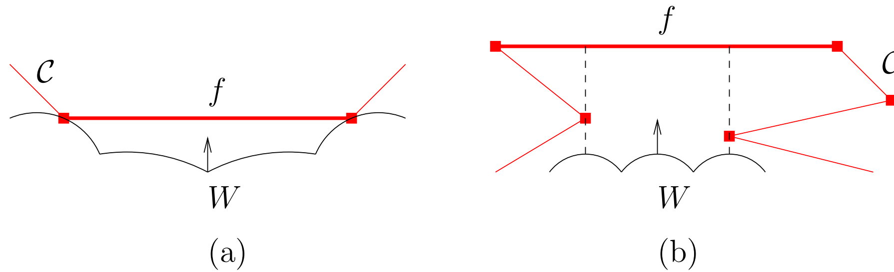

Figure 50: *Reaching segments f of $\mathcal{C}$: (a) Both endpoints of f are reached by W before f is covered by W. (b) W actually reaches only a portion of f (between the two dashed lines), because of visibility constraints.*

The algorithm does not necessarily detect the first time t_f when f is reached by W. Instead, we detect a time t'_f, when *some* path encoded in some wave of W reaches f. However, in order to estimate the time when f is completely covered by W correctly (although somewhat conservatively), the algorithm sets $t_{\text{stop}}(f) := t'_f + |f|$. We show below that t'_f is greater than t_f by at most $|f|$, hence the total gap between the time when f is first reached by W, and the time when the algorithm ascertains that f is completely covered, is at most $2|f|$.

Consider W', the sub-wavefront of W that covers a segment f of $\mathcal{C}$. If f is a transparent edge, the well-covering property of f ensures that during these $2|f|$ simulation time units (since t_f) no wave of W' has reached "too far" beyond f. That is, all the bisector events of W' beyond f that have been detected and processed before $t_{\text{stop}}(f)$ occur in $O(1)$ cells near c (see Figure 46 for an illustration). This invariant is crucial for the time complexity of the algorithm, as it implies that no bisector event is detected more than $O(1)$ times — see below. If f is a contact interval, the paths encoded in W that reach f in our propagation do not reach f in the real SPM(s), by Corollary 3.18; therefore these paths do not leave c

(as shortest paths), and need not be encoded in the one-sided wavefronts that leave c. This property is also used below in the time complexity analysis of the algorithm.

Processing candidate bisector events. As long as the simulation clock has not yet reached $t_{\text{stop}}(W)$, at each step of the simulation we extract from the priority queue of W the candidate bisector event which involves the generator s_i with the minimum priority in the queue, and process it according to the high-level description in Section 4.3, the details of which are given next. Let x denote the unfolded image of the location of the candidate event (the intersection point of the two bisectors of s_i), and denote by W' the constant-size sub-wavefront of W that encodes the paths involved in the event. If s_i is neither the first nor the last source image in W, then $W' = (s_{i-1}, s_i, s_{i+1})$. The generator s_i cannot be the only source image in W, since in this case its two bisectors would be rays emanating from s_i, and two such rays cannot intersect (beyond e). If s_i is either the first or the last source image in W, then W' is either (s_i, s_{i+1}) or (s_{i-1}, s_i), respectively. Denote by π_1 (resp., π_2) the path from the first (resp., last) source image of W' to x, or, more precisely, the respective unfolded straight segments of (common) length *priority*(s_i).

We use the tracing procedure defined above for each of the paths π_1, π_2. For any path π, denote by $\mathcal{C}(\pi)$ the first element of $\mathcal{C}$ (along π) that π intersects, if such a point exists. The following two cases can arise:

Case (i): The bisector event at x is *true with respect to the propagation of* W *in* $T_B(e)$ (see Figure 51(a)),[18] which means that none of π_1, π_2 intersects the boundary chain $\mathcal{C}$, and both paths are traced along a common block sequence in $T_B(e)$. (Recall that the unfolded blocks of $T_B(e)$ might overlap each other, so the latter condition is necessary to ensure that both paths reach x on the same layer of the Riemann structure; see Figure 51(b) for a counterexample.) By definition of a block tree, this is a necessary and *sufficient* condition for the event to be true (with respect to the propagation of W in $T_B(e)$); however, a following simulation restart might still discard this candidate event, forcing the simulation to reach it again. If s_i is neither the first nor the last source image in W, we DELETE s_i from W, and recompute the priorities of its neighbors s_{i-1}, s_{i+1}, as follows. Since all the source images of W are currently unfolded to the same plane $\Lambda(W)$, we can compute, in constant time, the intersection point p, if it exists, of the new bisector $b(s_{i-1}, s_{i+1})$ (stored in the data structure during the DELETE operation) with the bisector of s_{i-1} that is not incident to x. If the two bisectors do not intersect each other (p does not exist), we put *priority*$(s_{i-1}) := +\infty$; otherwise *priority*(s_{i-1}) is the length of the straight line from s_{i-1} to p, ignoring any visibility constraints, or the possibility that the two bisectors reach p through different block sequences — see below. The priority of s_{i+1} is recomputed similarly.

If $s_i = s_1$ is the first but not the last source image in W, we DELETE s_1 from W (that is, s_2 becomes the first source image in W), and define the first (unfolded) bisector of W as a ray from s_2 through x; the priority of s_2 is recomputed as above. If s_i is the last but not the first source image in W, it is handled symmetrically.

Case (ii): The bisector event at x is *false with respect to the propagation of* W *in* $T_B(e)$:

[18] We remind the reader that the event may still be false in the actual map SPM(s).

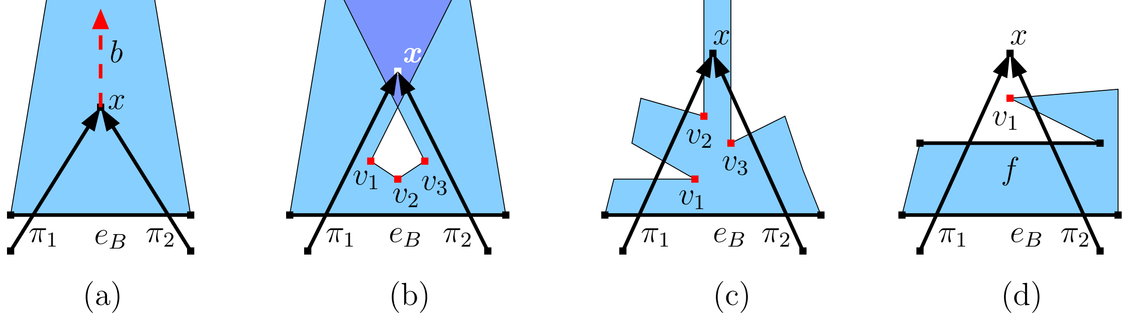

Figure 51: *Processing a candidate bisector event x. The outermost paths π_1, π_2 of W involved in x are drawn as thick arrows. In (a) x is a true bisector event; the new bisector b between the generators of π_1, π_2 is shown dashed. In (b–d) x is a false candidate. (b) π_1, π_2 do not intersect $\mathcal{C}$, but reach x through different layers of the Riemann structure that overlap each other (the region of overlap is darkly shaded). By Lemma 5.3, at least one vertex of $\mathcal{V} = \{v_1, v_2, v_3\}$ is visible from the portion of e_B between π_1 and π_2; the same is true in (c), where both π_1, π_2 intersect $\mathcal{C}$. (d) $\mathcal{C}(\pi_1) = \mathcal{C}(\pi_2)$ is a segment f of $\mathcal{C}$. No vertex of $\mathcal{V}$ (here $\mathcal{V} = \{v_1\}$) is visible from the portion of e_B between π_1 and π_2.*

Either at least one of the paths π_1, π_2 intersects $\mathcal{C}$, or π_1, π_2 are traced towards x along different block sequences in $T_B(e)$, reaching the location x in different layers of the Riemann structure that overlap at x. See Figure 51(b–d) for an illustration.

If π_1 intersects $\mathcal{C}$, denote the first such intersection point (along π_1) by z and the segment $\mathcal{C}(\pi_1)$, that contains z, by f (if z is a vertex of $\mathcal{C}$ and therefore is incident to two segments f, repeat this procedure for each such f). We compute z and update $t_{\text{stop}}(f) := \min\{t_{\text{stop}}(f), d_z + |f|\}$, where d_z is the distance from s to z along π_1. As described above, and with the visibility caveats noted there, the expression $d_z + |f|$ is a time at which W will certainly have swept over f. [19] We also update $t_{\text{stop}}(W) := \min\{t_{\text{stop}}(f), t_{\text{stop}}(W)\}$. If, as the result of this update, $t_{\text{stop}}(W)$ becomes less than or equal to the current simulation time, we conclude that f is already fully covered. We then stop the propagation of W and process f as a covered segment of $\mathcal{C}$ (as described below), immediately after completing the processing of the current bisector event. Note that in this case, that is, when $t_{\text{stop}}(f)$ gets updated because of the detection of the crossing of the wavefront of f at z, which causes $t_{\text{stop}}(W)$ to go below the current simulation clock t, we have $t_{\text{stop}}(W) = t_{\text{stop}}(f) = d_z + |f| \leq t = d_z + d(z, x)$, where $d(z, x)$ is the distance from z to x along π_1; see Figure 52. Hence $d(z, x) \geq |f|$. This however violates the invariant that we want to maintain, namely, that we only process bisector events that lie no farther than $|f|$ from an edge f of $\mathcal{C}$. Nevertheless, this can happen at most once per edge f, because from now on $t_{\text{stop}}(W)$ will not exceed $t_{\text{stop}}(f)$. We will use this property in the time complexity analysis below.

If π_2 intersects $\mathcal{C}$, we treat it similarly.

Regardless of whether π_1, π_2, or neither of them intersects $\mathcal{C}$, we then proceed as follows. Denote by τ the triangle bounded by the images of e, π_1 and π_2, unfolded to $\Lambda(W)$, and

[19] We could have used here instead of $|f|$ the expression $\max\{|za|, |zb|\}$, where a, b are the endpoints of f, but this optimization does not affect the asymptotic performance of the algorithm.

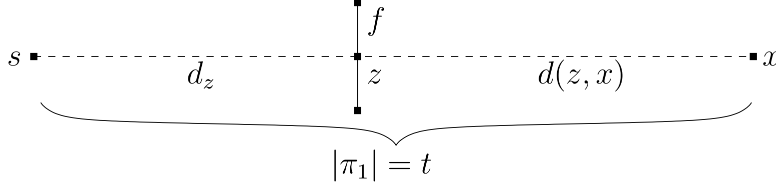

Figure 52: *If* $d_z + |f| \le t = d_z + d(z,x)$, *then* $d(z,x) \ge |f|$.

denote by $\mathcal{V}$ the set of the (at most $O(1)$) vertices of $\mathcal{C}$ that lie in the interior of τ. Since it takes $O(\log n)$ time to unfold each segment of $\mathcal{C}$, it takes $O(\log n)$ time to compute $\mathcal{V}$.

Assume first that π_1, π_2 satisfy the assumptions of Lemma 5.3; it follows that $\mathcal{V}$ is not empty (see Figure 51(b, c)). We trace the path from each generator in W' to each vertex v of $\mathcal{V}$, and compute *claimer*(v) (which satisfies $d(\mathit{claimer}(v), v) = \min\big\{\{d(s', v) \mid v \text{ is visible from } s' \in W'\} \cup \{+\infty\}\big\}$). Denote by u the vertex of $\mathcal{V}$ so that $t_u := d(\mathit{claimer}(u), u) = \min_{v \in \mathcal{V}} d(\mathit{claimer}(v), v)$; by Lemma 5.3, at least one vertex of $\mathcal{V}$ is visible from at least one generator in W', and therefore t_u is finite. As we will shortly show in Corollary 5.8, $t_u < t_x$ (where $t_x = \mathit{priority}(s_i)$ is the current simulation time). (This claim is "intuitively obvious" — see Figure 51(b–d), but does require a rigorous proof.) This implies that the propagation is invalid for $t \ge t_u$. We thus *restart* the propagation at time t_u, as follows.

Let W_u denote the last version of (the data structure of) W that has been computed before time t_u. We SPLIT W_u into sub-wavefronts W_1, W_2 at $s' := \mathit{claimer}(u)$ at the simulation time t_u, so that *range*(W_1) is the prefix of *range*(W_u) up to u, and *range*(W_2) is the rest of *range*(W_u) (to retrieve the range that is consistent with the version W_u we can simply store all the versions of *range*(W) — recall that each uses only constant space, because we can keep it unfolded). Discard all the later versions of W. We set $t_{\text{stop}}(W_1)$ (resp., $t_{\text{stop}}(W_2)$) to be the minimal $t_{\text{stop}}(f)$ value (as calculated before time t_u) among all segments f in *range*(W_1) (resp., *range*(W_2)). We replace the last (resp., first) unfolded bisector image of W_1 (resp., W_2) by the ray from s' through u, and correspondingly update the priority of s' in both new sub-wavefronts (recall from Section 5.1 that the SPLIT operation creates two distinct copies of s').

Assume next that the assumptions of Lemma 5.3 do not hold, which means that both π_1 and π_2 intersect $\mathcal{C}$, and that $\mathcal{C}(\pi_1) = \mathcal{C}(\pi_2)$, which is either a contact interval I or a transparent edge f of $\mathcal{C}$ (see Figure 51(d)). In the former case (a contact interval), the wave of s_i is not part of any sub-wavefront of W that leaves c (as shortest paths), and it is not involved in any further critical event inside c, as discussed above. To ignore s_i in the further simulation of the propagation of W in $T_B(e)$, we reset $\mathit{priority}(s_i) := +\infty$ (instead of deleting s_i from W, which would involve an unnecessary recomputation of the bisectors involving the neighbors of s_i). In the latter case, the following similar technical operation must be performed. Since s_i is a part of the resulting wavefront $W(e, f)$ (as will follow from the correctness of the bisector event processing, proved in Lemma 5.11 below), we do not want to delete s_i from W; yet, since s_i is not involved in any further critical event inside c, we want to ignore s_i in the further simulation of the propagation of W in $T_B(e)$ (that is, to ignore its priority in the priority queue), and therefore we update $\mathit{priority}(s_i) := +\infty$.

However, this artificial setting must be corrected later, when the propagation of W in $T_B(e)$ is finished, to ensure that the priority of s_i in $W(e, f)$ (which may be then merged into $W(f)$) is correctly set — we must then reset *priority*(s_i) to its true (current) value. We *mark* s_i to remember that its priority must be reset later.[20]

To summarize, in Case (i) we trace two paths and perform one DELETE operation and $O(1)$ priority queue operations, hence it takes $O(\log n)$ time to process a true bisector event. In Case (ii) we trace $O(1)$ paths, compute at most $O(1)$ unfolded images, and perform at most one SPLIT operation and $O(1)$ priority queue operations; hence it takes $O(\log n)$ time to process a false (candidate) bisector event. The correctness of the above procedure is established in Lemma 5.11 below, but first we describe the detection and the processing of the candidate vertex events that were not detected and processed during the handling of false candidate bisector events. This situation arises when the priority of the next event of W in the priority queue is equal or greater than $t_{\text{stop}}(W)$, in which case we stop processing the bisector events of W in $T_B(e)$, and proceed as described next.

Processing a covered segment of $\mathcal{C}$. Consider the situation in which the algorithm stops propagating W in $T_B(e)$ when the simulation reaches the time $t_{\text{stop}}(W) \neq +\infty$. We then must have $t_{\text{stop}}(W) = t_{\text{stop}}(f)$, for some segment f in *range*(W), so that all the currently known candidate events that involve the sub-wavefront of W that claims f have already been processed.

Another case in which the algorithm stops the propagation of W is when $t_{\text{stop}}(W) = +\infty$. This means that all the currently known candidate events of W have already been processed; that is, the former situation holds for each segment f in *range*(W). Therefore to treat the latter case we process each f in *range*(W), in the same manner as processing a single f in the former case, so we only consider the former situation.

Let f be such a segment of *range*(W). We compute the static wavefront $W(e, f)$ from the current dynamic wavefront W — if f is a transparent edge, then $W(e, f)$ is needed for the propagation process in further cells; otherwise (f is a contact interval) we do not need to compute $W(e, f)$ to propagate it further, but we need to know the extreme generators of $W(e, f)$ to ensure correctness of the simulation process, a step that will be explained in the proof of Lemma 5.11 below. Since the computation in the latter case is almost identical to the former, we treat both cases similarly (up to a single difference that is detailed below).

Since $f \in \mathcal{C}$ defines a unique homotopy class of paths from e_B to f within $T_B(e)$, the sub-wavefront of W that claims points of f is indeed a single contiguous sub-wavefront $W' \subseteq W$. We determine the *candidate* extreme claimers of f by performing SEARCH in W for each of the endpoints a, b of f (note that the candidates are not necessarily true, since SEARCH does not consider visibility constraints). If the candidate claimer of a does not exist, we denote by a' the closest to a point of f intersected by an extreme bisector of W – see Figure 53(a). (If there is no such a', we can already determine that W claims no points on f, and no

[20] In the previously described case, where $\mathcal{V}$ contains vertices that are visible from W', the above technical procedure is not currently needed, because we will first process vertex events at some of the vertices of $\mathcal{V}$, which will cause restarts of the simulation, involving splitting W at the appropriate claimers. These restarts may eventually lead to situations with $\mathcal{V} = \emptyset$, which will then be processed as described above.

further processing of f is needed.[21]) Symmetrically, we SEARCH for the claimer of b, and, if it is not found, we define b' similarly. If a (resp., b) is claimed by W, denote by π_1 (resp., π_2) the path $\pi(claimer(a), a)$ (resp., $\pi(claimer(b), b)$); otherwise denote by π_1 (resp., π_2) the path $\pi(claimer(a'), a')$ (resp., $\pi(claimer(b'), b')$). Note that π_1, π_2 might intersect $\mathcal{C}$; as we will shortly see, this is exactly the situation in which we can detect a vertex event that has occurred earlier but has not yet been detected. Denote by W' the sub-wavefront of W between the generators of π_1 and π_2 (inclusive), and use π_1, π_2 to define (and compute) $\mathcal{V}$ as in the processing of a candidate bisector event (described above).

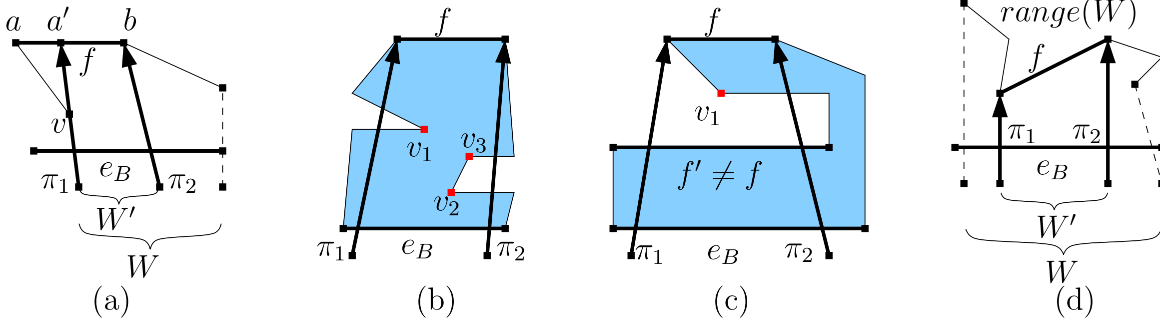

Figure 53: *Processing a covered segment f of range(W). (a) The endpoint a of f is not claimed by W, and π_1 is the shortest path to the last claimed point a'; the generator of π_1 is extreme in W (which has already been split at v) and π_1 lies on the extreme bisector of W. (b) By Lemma 5.3, at least one vertex of $\mathcal{V} = \{v_1, v_2, v_3\}$ is visible from the portion of e_B between π_1 and π_2. (c) $\mathcal{C}(\pi_1) = \mathcal{C}(\pi_2)$ is a segment $f' \neq f$ of $\mathcal{C}$; f is not reached by W at all. No vertex of $\mathcal{V}$ is visible from the portion of e_B between π_1 and π_2. (d) Since $d_f = |\pi_1| < |\pi_1|$, W is split at the generator of π_1 first.*

Assume first that π_1, π_2 satisfy the assumptions of Lemma 5.3. It follows that $\mathcal{V}$ is not empty, and at least one vertex of $\mathcal{V}$ is visible from (all generators in W', and, in particular, from) its claimer in W' (see, e.g., Figure 53(b)). Then the case is processed as Case (ii) of a candidate bisector event (described above), with the following difference: Instead of tracing a path from each source image in W' to each vertex $v \in \mathcal{V}$ (which is too expensive now, since W' may have non-constant size), we first SEARCH in W' for the claimer of each such v and then trace only the path $\pi(claimer(v), v)$. (Then we restart the simulation from the earliest time when a vertex v of $\mathcal{V}$ is reached by W, splitting W at $claimer(v)$.)

Assume next that the assumptions of Lemma 5.3 do not hold, which means that both π_1 and π_2 intersect $\mathcal{C}$, and that $\mathcal{C}(\pi_1) = \mathcal{C}(\pi_2)$, which is either f or a segment $f' \neq f$ of $\mathcal{C}$. In the latter case, since f is not reached by W at all, no further processing of f is needed (see Figure 53(c)) — we delete f from $range(W)$, and update $t_{\text{stop}}(W) := \min\{t_{\text{stop}}(f') | f' \in range(W) \setminus \{f\}\}$. In the former case, if both π_1, π_2 are extreme in W, then we have $W' = W$; the further processing of f is described below. Otherwise (at least one of π_1, π_2 is not extreme in W), we first have to split W, as follows. If π_1 and π_2 are not extreme in W, denote by d_f the minimum of $|\pi_1|, |\pi_2|$; if only one path $\pi \in \{\pi_1, \pi_2\}$ is non-extreme in W, let $d_f := |\pi|$. Without loss of generality, assume that $d_f = |\pi_1|$ — see Figure 53(d). We restart the

[21]Note that this situation may only arise if $t_{\text{stop}}(W) = +\infty$ and f is not the only segment in $range(W)$.

simulation from time $|\pi_1|$, splitting W at the generator of π_1, as described in Case (ii) of the processing of a candidate bisector event.

It is only left to describe the case where $W' = W$ and f is the only segment of $range(W)$. If f is a contact interval, no further processing of f is needed. Otherwise (f is a transparent edge), we have to make the following final updates (to prepare $W(e, f)$ for the subsequent merging procedure at f and for further propagation into other cells). First, we recalculate the priority of each *marked* source image (recall that it was temporarily set to $+\infty$), and update the priority queue component of the data structure accordingly. Next, we update the source unfolding data (and $\Lambda(W)$), as follows. Let $\mathcal{B}$ be the block sequence traversed by W from e to f along $T_B(e)$, including (resp., excluding) B if the first (resp., last) facet of B lies on $\Lambda(W)$, and let $\mathcal{E}$ be the edge sequence associated with $\mathcal{B}$. We compute the unfolding transformation $U_{\mathcal{E}}$, by composing the unfolding transformations of the $O(1)$ blocks of $\mathcal{B}$. We update the data structure of $W(e, f)$ to add $U_{\mathcal{E}}$ to the unfolding data of all the source images in $W(e, f)$, as described in Section 5.1. As a result, for each generator s_i of $W(e, f)$, the polytope edge sequence $\mathcal{E}_i$ is the concatenation of its previous value with $\mathcal{E}$, and all the generators in $W(e, f)$ are unfolded to the plane of an extreme facet incident to f.

Remark: As described in Section 4.2, the basic operation performed in the *merging* procedure at a transparent edge f (which results in a pair of one-sided wavefronts at f) is the computation of a bisector between two generators in two distinct wavefronts that reach (a fixed side of) f. Note that the above invariant, that all the generators in such a wavefront $W(e, f)$ are ready to be unfolded to a plane of a facet intersected by f (that is, for each generator in $W(e, f)$ this unfolding transformation is available by traversing its path in the tree that stores $W(e, f)$, in $O(\log n)$ time), allows us to compute a bisector between two generators in two distinct wavefronts that reach f, in $O(\log n)$ time. We omit further (simple) details of this operation.

To summarize, we trace $O(1)$ paths and perform one SPLIT and at most $O(1)$ SEARCH operations, for each of at most $O(1)$ segments of $\mathcal{C}$. Then we perform at most one source unfolding information update for each transparent edge in $\mathcal{C}$. All these operations take a total of $O(\log n)$ time. However, we also perform a single priority update operation for each marked generator which has participated in a candidate bisector event beyond a transparent edge of $\mathcal{C}$. A linear upper bound on the total number of these generators, as well as the number of the processed candidate events, is established next.

5.3.2 Correctness and complexity analysis.

We start by observing, in the following lemma, a very basic property of W that asserts, informally, that distances from generators *increase* along their bisectors; this will be used in the correctness analysis of the simulation algorithm.

Lemma 5.4. *Let $s_i, s_j \in W$ be a pair of generators that become neighbors at a bisector event x during the propagation of W through $T_B(e)$, where an intermediate generator s' gets eliminated. Then (i) the portion of the bisector $b(s_i, s_j)$ that is closer to s' than x is claimed, among s_i, s' and s_j, by s', and (ii) x is closer to s_i (and s_j) than any other point on the portion of $b(s_i, s_j)$ that is not claimed by s'.*

Proof: (i) Assume the contrary. Then, as is easily seen, the region $R(s')$ claimed by s' must fully lie on one side of $b(s_i, s_j)$ (on $\Lambda(W)$); without loss of generality, assume that $R(s')$ lies entirely on the same side of $b(s_i, s_j)$ as s_i (so that $R(s') \cap b(s_i, s_j) = x$; s_j lies on the other side of $b(s_i, s_j)$). See Figure 54(a) for an illustration. Consider the straight segment $\overline{s_i s_j}$ (on $\Lambda(W)$), and let $r = \overline{s_i s_j} \cap b(s_i, s_j)$, $q = \overline{s_i s_j} \cap b(s', s_j)$ and $u = \overline{s_i s_j} \cap b(s_i, s')$ be the three intersection points of $\overline{s_i s_j}$ with the corresponding bisectors. Now consider the straight segments $\overline{s'q}$ and $\overline{s'u}$ on $\Lambda(W)$: Since $q \in b(s', s_j)$, we have $|\overline{s'q}| = |\overline{s_j q}|$; similarly, $|\overline{s_i r}| = |\overline{s_j r}|$ and $|\overline{s_i u}| = |\overline{s'u}|$. Since $|\overline{s_j q}| > |\overline{s_j r}|$ and $|\overline{s_i r}| > |\overline{s_i u}| + |\overline{uq}|$, we have $|\overline{s'q}| > |\overline{s'u}| + |\overline{uq}|$, which contradicts the triangle inequality.

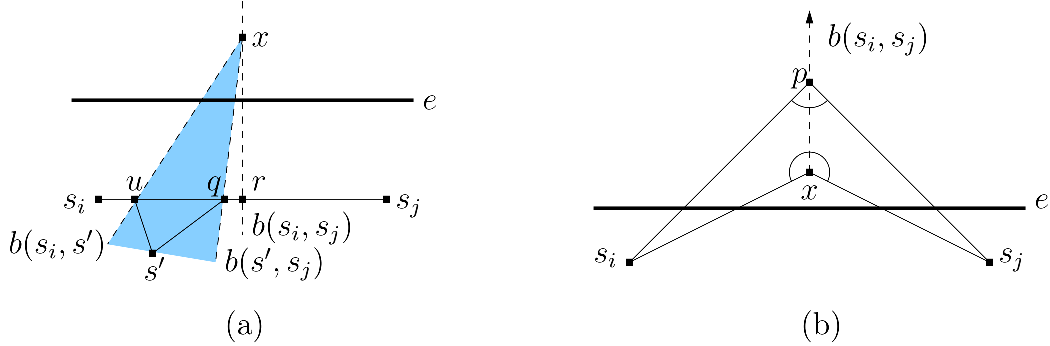

Figure 54: *(a) The impossible situation where the region $R(s')$ claimed by s' (shaded) lies entirely on one side of $b(s_i, s_j)$. (b) s_i and s_j must lie on the same side of the line l that supports e, and x and p must lie on the other side of l.*

(ii) Assume the contrary: Let p be a point on the portion of the bisector $b(s_i, s_j)$ that is not claimed by s', so that $d(s_i, p) \leq d(s_i, x)$. Then $\angle s_i p x \geq \angle s_i x p$, and, since p and x are equidistant from s_i and from s_j, $\angle s_i p x = \angle s_j p x$ and $\angle s_i x p = \angle s_j x p$. However, since both s_i and s_j are in the one-sided wavefront $W(e_B)$, s_i and s_j must lie on the same side of the line l that supports e_B, and x and p must lie on the other side of l (see Figure 54(b)). Hence $\angle s_i x s_j > \pi > \angle s_i p s_j$, a contradiction. □

Lemma 5.5. *Assume that all bisector events of W that have occurred up to some time t have been correctly processed, and the data structure of W has been correctly updated. Let p be a point* tentatively *claimed by a generator $s_i \in W$ at time $d(s_i, p) \leq t$, meaning that the claim is only with respect to the* current *generators in W (at time t), and where we ignore any visibility constraints of $\mathcal{C}$. Denote by $R(s_i)$ the unfolded region, that is enclosed between the bisectors of s_i currently stored in the data structure. Then $p \in R(s_i)$, and $p \notin R(s_j)$, for any other generator $s_j \neq s_i$ in W.*

Proof: The claim that $p \in R(s_i)$ is trivial, since the bisectors of s_i that are currently stored in the data structure have been computed before time t, and are therefore correct, by assumption.

For the second claim, assume to the contrary that there exists a generator $s_j \neq s_i$ in W so that $p \in R(s_j)$. Denote by q the first point along $\pi(s_j, p)$ that is equally closest to s_j and to some other generator $s' \in W$ (such q and s' must exist, since $d(s_i, p) < d(s_j, p)$); that is,

$q = \pi(s_j, p) \cap b(s', s_j)$. The fact that in the data structure p lies in $R(s_j)$ means that the bisector $b(s', s_j)$ is not correctly stored in the data structure, and thus it cannot be a part of $W(e_B)$; therefore $b(s', s_j)$ emanates from a bisector event location x that lies within c. By Lemma 5.4, $d(s', x) < d(s', q) < d(s', p) \le t$; hence, the bisector event when $b(s', s_j)$ is computed occurs before time t, and therefore $b(s', s_j)$ is correctly stored in the data structure — a contradiction. □

In particular, Lemma 5.5 shows that when a vertex event at v is discovered during the processing of another event at simulation time t, or is processed since a segment of $\mathcal{C}$ that is incident to v is covered at time t, the tentative claimer of v (among all the current generators in W) is correctly computed, assuming that all bisector events of W that have occurred up to time t have been correctly processed. We will use this argument in Lemma 5.11 below.

Lemma 5.6. *Assume that all bisector events of W that have occurred up to some time t have been correctly processed, and the data structure of W has been correctly updated at all these events. If two waves of a common topologically constrained portion of W are adjacent at t, then their generators must be adjacent in the generator list of W at time t.*

Proof: Assume the contrary. Then there must be two source images s_i, s_j in a common topologically constrained portion $W' \subseteq W$ such that their respective waves w_i, w_j are adjacent at some point x at time t (that is, $d(s_i, x) = d(s_j, x) = t \le d(s_k, x)$ for all other generators s_k in W), but there is a positive number of source images $s_{i+1}, \ldots, s_{j-1}$ in the generator list of W' at time t between s_i and s_j, whose distances to x are necessarily larger than $d(s_i, x)$ (and their waves in W' at time t are nontrivial arcs). See Figure 55 for an illustration.

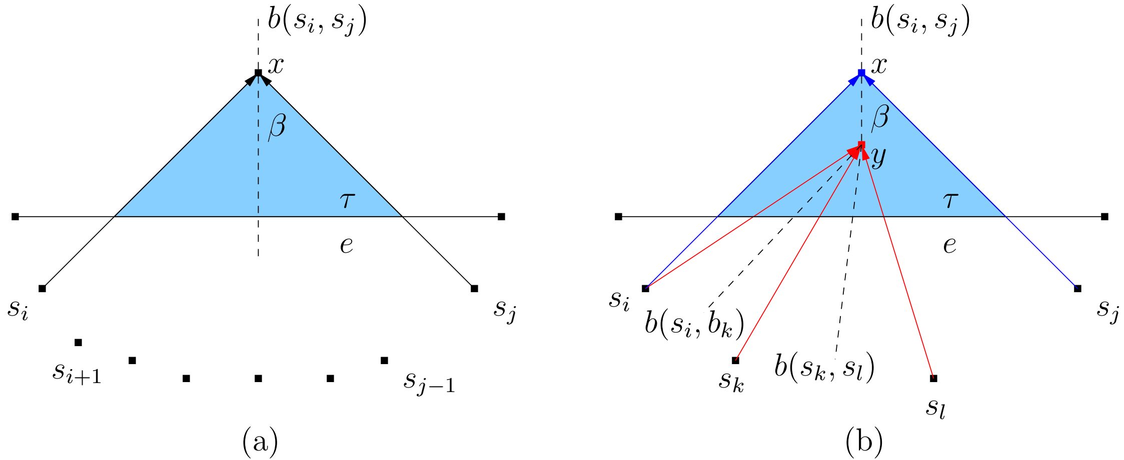

Figure 55: *The waves from the source images s_i, s_j collide at x. Each of the two following cases contradicts the assumption in the proof of Lemma 5.6: (a) The portion β of $b(s_i, s_j)$ intersects the transparent edge e; (b) The generator s_k is eliminated at time $t_y = d(s_i, y) < d(s_i, x) = t$.*

Consider the situation at time t. Since w_i, w_j belong to a common topologically constrained W', it follows that e, $\pi(s_i, x)$ and $\pi(s_j, x)$ unfold to form a triangle τ in an unfolded block sequence of $T_B(e)$ (so that τ is not intersected by $\mathcal{C}$).

Consider the "unfolded" Voronoi diagram $\mathrm{Vor}(\{s_i, \ldots, s_j\})$ within τ. By assumption, x lies in the Voronoi cells $V(s_i), V(s_j)$ of s_i, s_j, respectively, separated by a Voronoi edge β, which is a portion of $b(s_i, s_j)$. If β intersects e (see Figure 55(a)), then s_i and s_j claim consecutive portions of e in $W(e)$, so s_i and s_j must be consecutive in W already at the beginning of its propagation within $T_B(e)$, a contradiction.

Otherwise, β ends at a Voronoi vertex within τ — see Figure 55(b). Thus both cells $V(s_i), V(s_j)$ have Voronoi vertices in τ, and we select that vertex y whose distance t_y to its claimers is the smallest. Clearly, y is the location of the bisector event in which some generator $s_k \in W$ is eliminated; moreover, s_k must belong to W', by construction. By Lemma 5.4, $t_y < t$, and therefore, by our assumption, the bisector event at y has been correctly processed, so s_i and s_j must be consecutive in W already before time t — a contradiction.□

Lemma 5.6 shows that if all the events considered by the algorithm are processed correctly, then all the true bisector events of the first kind are processed by the algorithm, since, as the lemma shows, such events occur only between consecutive generators of W. Let W' be a topologically constrained portion of W, and denote by $R(W', t)$ the region within $T_B(e)$ that is covered by W' from the beginning of the simulation in $T_B(e)$ up to time t. By definition of the topologically constrained wavefront, $\partial R(W', t)$ consists only of e_B and of the unfolded images of the waves and of the extreme bisectors of W'. Another role of Lemma 5.6 is in the proof of the following observation.

Corollary 5.7. *$R(W', t)$ is not punctured (by points or "islands" that are not covered by W' at time t).*

Proof: Follows directly from Lemma 5.6, since, otherwise, $R(W', t)$ would have to contain a point q where a pair of waves, generated by the respective generators s_i, s_j, collide, and e_B and the paths $\pi(s_i, q), \pi(s_j, q)$ enclose an island within the (unfolded) triangle that they form. This however contradicts the proof of Lemma 5.6. □

Corollary 5.8. *When a vertex event at v is discovered during the processing of a candidate event at simulation time t (either a bisector event x or an event involving a covered segment f of $\mathcal{C}$), the vertex v is reached by W no later than time t.*

Proof: Indeed, by the way vertex events are discovered, v must lie in an unfolded triangle τ formed as in the proof of Lemma 5.6, where the waves of the respective generators s_i, s_j either collide at a bisector event location x, or are adjacent in the wavefront that covers the segment f of $\mathcal{C}$. Since the two sides of τ incident to x belong to $R(W', t)$, for some topologically constrained portion W' of W that contains s_i, s_j, Corollary 5.7 implies that all of τ is contained in $R(W', t)$, which implies the claim. □

The analysis of the simulation restarts. Before we show the correctness of the processing of the true critical events, let us discuss the processing of the false candidates. First, note that the simulation can be aborted at time t' (during the processing of a false candidate event) and restarted from an earlier time $t'' < t'$ only if there exists some true vertex event

x that should occur at time $t \leq t''$ and has not been detected until time t' (in the aborted version of the simulation). Note that whenever a false candidate event $x' \notin \{x_1, \ldots, x_m\}$ is processed at time t', one of the three following situations must arise.

(i) It might be the case that x' is not currently (at time t') determined to be false, since both paths involved in x' are traced along the same block sequence and do not intersect $\mathcal{C}$, x' is false "just" because there is some earlier true vertex event that is still undetected. Then a new (invalid) version of W corresponding to time t' is created at time t'.

Otherwise, x' is immediately determined to be false (since either one of the involved paths intersects $\mathcal{C}$ or the paths are traced along different block sequences). In this case either (ii) an earlier candidate vertex event x'' (occurring at some time $t'' < t'$) is currently detected and the simulation is restarted from t'', or (iii) x' is a bisector event which occurs outside $T_B(e)$, so it involves only bisectors that do not participate in any further critical event inside $T_B(e)$. In this case a new version of W, corresponding to the time t', is created, in which the generator that is eliminated at x' is marked and its priority is set to $+\infty$.

In any of the above cases, none of the existing true (valid) versions of W is altered (although some invalid versions may be discarded during a restart); moreover, a new invalid version corresponding to time t' may be created (without restarting the simulation yet) only if there is some true event that occurred at time $t < t'$ but is still undiscovered at time t'.

Order the $O(1)$ vertices of $\mathcal{C}$ that are reached by W (that is, the locations of the true vertex events) as $v_1, \ldots, v_m$, where W reaches v_1 first, then v_2, and so on (this is not necessarily their order along $\mathcal{C}$); denote by t_j, for $1 \leq j \leq m$, the time at which W reaches v_j. Assume that if the simulation is restarted because of a vertex event at v_j, then the simulation is restarted exactly from time t_j — we show that this assumption is correct in Lemma 5.11 below; in other words, the simulation is only restarted from times $t_1, \ldots, t_m$.

Lemma 5.9. *When the vertex events at vertices $v_1, \ldots, v_k$, for $1 \leq k \leq m$, are already detected and processed by the algorithm, the simulation is never restarted from time t_k or earlier.*

Proof: Since the simulation restart from time t discards all existing versions of W that correspond to times $t' > t$, the claim of the lemma is equivalent to the claim that all the versions of W that were created at time t_k or earlier will never be discarded by the algorithm if all the vertex events at vertices $v_1, \ldots, v_k$ have already been detected and processed. We prove the latter claim by induction on k.

For $k = 1$, the version of W created at time t_1 can only be discarded if a vertex event which occurs earlier than t_1 is discovered, which is impossible since v_1 is the first vertex reached by W.

Now assume that the claim is true for $v_1, \ldots, v_{k-1}$, and consider the version W_k of W that is created at time t_k when the vertex events at vertices $v_1, \ldots, v_k$ are already detected and processed. The algorithm may discard W_k only when at some time $t' > t_k$ a vertex v is discovered, such that v is reached by W at time $t_v < t_k$, and therefore v must be in $\{v_1, \ldots, v_{k-1}\}$. This therefore cannot happen, since v has already been processed at time $t_v < t'$, and W, that already encodes the split at the claimer of v, cannot detect and process

the event at v again without discarding the version that corresponds to time $t_1 \leq t_v \leq t_{k-1}$ (and discarding this version contradicts the induction hypothesis). □

Lemma 5.10. *For each $1 \leq j \leq m$, the simulation is restarted from time t_j at most 2^{j-1} times.*

Proof: By induction on j. By Lemma 5.9, the simulation is restarted from time t_1 at most once. Now assume that $j \geq 2$ and that the claim is true for times $t_1, \ldots, t_{j-1}$.

By Lemma 5.11, the vertex event at v_j is eventually processed at time t_j; by Lemma 5.9, there are no further restarts from time t_j after we get a version of W that encodes all the events at $v_1, \ldots, v_j$. Hence the simulation may be restarted from time t_j only once each time that W ceases to encode the vertex event at v_j, and this may only happen either at the beginning of the simulation, or when the simulation is restarted from a time earlier than t_j. Since, by the induction hypothesis, the simulation is restarted from times $t_1, \ldots, t_{j-1}$ at most $\sum_{i=1}^{j-1} 2^{i-1} = 2^{j-1} - 1$ times, the simulation may be restarted from time t_j at most 2^{j-1} times. □

Correctness of true critical event processing. We are now ready to establish the correctness of the simulation algorithm. Since this is the last remaining piece of the inductive proof of the whole Dijkstra-style propagation (Lemmas 4.2 and 4.5), we may assume that all the wavefronts were correctly propagated to some transparent edge e, and consider the step of propagating from e. This implies that $W(e_B)$ encodes all the shortest paths from s to the points of e_B from one fixed side. Now, let $x_1, \ldots, x_m$ be all the *true critical events* (that is, both bisector and vertex events that are true with respect to the propagation of W in $T_B(e)$), ordered according to the times $t_1, \ldots, t_m$ at which the locations of these events are first reached by W. Since we assume general position, $t_1 < \cdots < t_m$.

Assume now that at the simulation time t_k (for $1 \leq k \leq m$) all the true events that occur before time t_k have been correctly processed; that is, for each such bisector event x_i, the corresponding generator has been eliminated from W at simulation time t_i, and for each such vertex event x_j, W has been split at simulation time t_j at the generator that claims the corresponding vertex. Note that the assumption is true for simulation time t_1, since the processing of false candidate events does not alter $W(e_B)$ (which represents the wavefront before any event within $T_B(e)$; its validity follows from the inductive correctness of the merging procedure and is not violated by the processing of false events).

Lemma 5.11. *Assuming the above inductive hypothesis, the next true critical event x_k is correctly processed at simulation time t_k, possibly after a constant number of times that the simulation clock has reached and passed t_k (to process a later false candidate event) without detecting x_k, each time resulting in a simulation restart.*

Proof: There are two possible cases. In the first case, x_k is a true bisector event, in which the wave of a generator s' in W is eliminated by its neighbors at propagation time t_k. The only condition needed for the processing of x_k at time t_k is that *priority*(s') at time t_k must be equal to t_k. Any possible false candidate event that is processed before x_k and after the

processing of all true events that take place before time t_k, may only create new invalid versions that correspond to times which are later than time t_k (since a false candidate event can arise only when an earlier true event is still undetected). This implies that s' has not been deleted from any valid version of W that corresponds to time t_k or earlier, and all such valid versions exist. By this fact and by the inductive hypothesis, the bisectors of s' have been computed correctly either already in $W(e_B)$, or during the processing of critical events that took place before time t_k. By definition, $priority(s')$ is the distance from s' to the event point, and is thus equal to t_k, so the above condition is fulfilled.

In the second case, x_k is a true vertex event that takes place at a vertex $v \in \mathcal{C}$, which is claimed by some generator s_v in W. By the argument used in the first case, s_v has not been deleted from W at an earlier (than t_k) simulation time, and each point on the path $\pi(s_v, v)$ is claimed by s_v at time t_k or earlier. Therefore s_v can only be deleted from a version of W at time later than t_k when a false bisector event involving s_v is processed. Moreover, a sub-wavefront including s_v can be split from a version of W at time later than t_k (and v can be removed from $range(W)$) when a false vertex event is processed. We show next that in both cases, x_k is detected and the simulation is restarted from time t_k, causing x_k to be processed correctly.

Consider first the case where s_v is not deleted in any later false candidate event. In that case, when we stop the propagation of W, v is in $range(W)$, and therefore at least one segment f of $range(W)$ that is incident to v is ascertained to be covered at that time. Since s_v is in W, Lemma 5.5 implies that the result of the SEARCH procedure that the algorithm uses to compute the claimer of v is s_v, and, by Corollary 5.2, the tracing procedure correctly computes $d(s_v, v)$ to be t_k. Since x_k is the next true vertex event, the distance from the other endpoint of f to its claimer is larger than or equal to t_k, and, since W has not yet been split at v, $\pi(s_v, v)$ is not an extreme bisector of W. Hence the algorithm sets $d_f := t_k$, and W is split at s_v at simulation time t_k, as required.

Consider next the case where s_v is deleted (or split) from W at a false event x' at time $t' \geq t_k$. Suppose first that x' is a false bisector event. Then v must lie in the interior of the region τ bounded by e and by the paths to the location of x' from the outermost generators of W involved in x'. The algorithm traces the paths to v and to (some of) the other vertices of $\mathcal{C}$ in τ from all the generators of W that are involved in x', including s_v (see Figure 51(b, c)); then all such distances are compared. Only distances from each such generator s' to each vertex that is visible from s' (within the unfolded blocks of $T_B(e)$) are taken into account, since, by Corollary 5.2, all visibility constraints are detected by the tracing procedure. The vertex v must be visible from s_v and the distance $d(s_v, v)$ must be the shortest among all compared distances, since, by the inductive hypothesis, all vertex events that are earlier than x_k have already been processed (and W has already been split at these events). By Lemma 5.5 and by Corollary 5.2, the tentative claimer (among all current generators in W) of each vertex u is computed correctly. No generator s' that has already been eliminated from W can be closer to u than the computed $claimer(u)$, since, by Corollary 5.8 and by the inductive hypothesis, u would have been detected as a vertex event no later than the bisector event of s', which is assumed to have been correctly processed. Therefore the distance $d(claimer(u), u)$ is correctly computed for each such vertex u (including v), and therefore the

distance $d(s_v, v) = t_k$ is determined to be the shortest among all such distances. Hence the simulation is restarted from time t_k, and W is split at s_v at simulation time t_k, as asserted.

Otherwise, x' is a false vertex event processed when a segment f of $\mathcal{C}$ is ascertained to be fully covered by W, and v must lie in the interior of the region τ bounded by e, f, and by the paths from the outermost generators of W claiming f to the extreme points of f that are tentatively claimed by W (see Figure 53(b)). The algorithm performs the SEARCH operation in the sub-wavefront $W' \subseteq W$ that claims f for v and for all the other vertices of $\mathcal{C}$ in τ, and then compares the distances $d(\mathit{claimer}(u), u)$, for each such vertex u that is visible from its claimer (including v). By the same arguments as in the previous case, the distance $d(s_v, v) = t_k$ is determined to be the shortest among all such distances, the simulation is restarted from time t_k, and W is split at s_v at simulation time t_k, as asserted. $\square$

The above lemma completes the proof of the correctness of our algorithm, because it shows, using induction, that every true event will eventually be detected.

Remark: From a practical point of view, the algorithm can be greatly optimized, by using the information computed before the restart to speed-up the simulation after it is restarted. Moreover, we suspect that, in practice, the number of restarts that the algorithm will perform will be very small, significantly smaller than the bounds in the lemma.

Complexity analysis. By Lemma 5.10, the algorithm processes only $O(1)$ candidate vertex events (within a fixed $T_B(e)$), and, since the simulation is restarted only at a vertex event, it follows that each bisector event has at most $O(1)$ "identical copies", which are the same event, processed at the same location (and at the same simulation time) after different simulation restarts; at most one of these copies remains encoded in valid versions of W (and the rest are discarded). Hence for the purpose of further asymptotic time complexity analysis, it suffices to bound the number of the processed candidate bisector events that take place at different locations.

Note that each candidate bisector event x processed by the propagation algorithm falls into one of the five following types:

(i) x is a true bisector event.

(ii) x is a false candidate bisector event, during the processing of which an earlier-reached vertex of $\mathcal{C}$ has been discovered, and the simulation has been restarted.

(iii) x is a false candidate bisector event of a generator $s' \in W$, so that all paths in the wave from s' cross a common contact interval of $\mathcal{C}$ (a "dead-end") before the wave is eliminated at x.

(iv) x is a false candidate bisector event of a generator $s' \in W$, so that all paths in the wave from s' cross a common transparent edge f of $\mathcal{C}$ before the wave is eliminated at x, and the distance from f to x along $d(s', x)$ is greater than $2\,|f|$.

(v) x is a false candidate bisector event, as in (iv), except that the distance from f to x along $d(s', x)$ is at most $2\,|f|$.

We first bound the number of true bisector events (type (i)).

Lemma 5.12. *The total number of processed true bisector events, during the whole wavefront propagation phase, is $O(n)$.*

Proof: First we bound the total number of waves that are created (and propagated) by the algorithm.

The wavefront W is always propagated from some transparent edge e, within the blocks of a tree $T_B(e)$, for some block B incident to e, in the Riemann structure $\mathcal{T}(e)$ of e. A wave of W is split during the propagation only when W reaches a vertex of $\mathcal{C}$, the corresponding boundary chain of $T_B(e)$. Each such vertex is reached at most once (ignoring restarts) by each topologically constrained wavefront that is propagated in $T_B(e)$. There are only $O(1)$ such wavefronts, since there are only $O(1)$ paths in $T_B(e)$ (and corresponding homotopy classes). Each (side of a) transparent edge e is processed exactly once (as the starting edge of the propagation within its well-covering region), by Lemma 4.2, and e may belong to at most $O(1)$ well-covering regions of other transparent edges, that may use e at an intermediate step of their propagation procedures. There are $O(1)$ vertices in any boundary chain $\mathcal{C}$, hence at most $O(1)$ wavefront splits can occur within $T_B(e)$ during the propagation of a single wavefront. Since there are only $O(n)$ transparent edges e in the surface subdivision, and there are only $O(1)$ trees $T_B(e)$ for each e, we process at most $O(n)$ such split events. (Recall from Lemma 5.10 that a split at a vertex is processed at most $O(1)$ times.) Since a new wave is added to the wavefront only when a split occurs, at most $O(n)$ waves are created and propagated by the algorithm.

In each true bisector event processed by our algorithm, an existing wave is eliminated (by its two adjacent waves). Since a wave can be eliminated exactly once and only after it was earlier added to the wavefront, we process at most $O(n)$ true bisector events. □

Lemma 5.13. *The algorithm processes only $O(n)$ candidate bisector events during the whole wavefront propagation phase.*

Proof: There are at most $O(n)$ events of type (i) during the whole algorithm, by Lemma 5.12. By Lemma 5.10, there are only $O(1)$ candidate events of type (ii) that arise during the propagation of W in any single block tree $T_B(e)$. Since a candidate event of type (iii), within a fixed surface cell c, involves at least one wave that encodes paths that enter c through e_B but never leave c (that is, they traverse a facet sequence that contains a loop, and are therefore known not to be shortest paths beyond some contact interval in the loop), the total number of these candidate events during the whole propagation is bounded by the total number of generated waves, which is $O(n)$ by the proof of Lemma 5.12.

When a candidate event of type (iv) occurs at a location x at time t_x, while processing some fixed $T_B(e)$, consider the transparent edge f of $\mathcal{C}$ that is crossed by the wave from the generator s' eliminated at x. Denote by d_1 the distance from s' to f along $\pi(s', x)$, and denote by d_2 the distance along $\pi(s', x)$ from f to x; that is, $d_2 > 2\,|f|$ and $d_1 + d_2 = d(s', x) = t_x$. Before the update, the value of $t_{\text{stop}}(f)$ must have been equal or greater than $t_x > d_1 + 2\,|f|$, since otherwise f would have been ascertained to be covered before time t_x, and therefore the event at t_x would have not be processed; hence, after the update, we have $t_{\text{stop}}(f) = d_1 + |f| < t_x$. Therefore, immediately after the processing of the event at t_x we

detect that f has been covered; by Lemma 5.10 each f is detected to be covered at most $O(1)$ times, and, since there are only $O(1)$ transparent edges in $\mathcal{C}$, there are at most $O(1)$ events of type (iv) during the propagation of W in $T_B(e)$.

Consider now a candidate event of type (v) that occurs at a location x at time t_x after crossing the transparent edge f of $\mathcal{C}$. This event may also be detected during the propagation of the wavefront through f into further cells, and therefore it must be counted more than once during the whole wavefront propagation phase. However, on $\Lambda(W)$, x lies no further than $2\,|f|$ from the image of f, and therefore the shortest-path distance from f to the location of x on ∂P cannot be greater than $2\,|f|$; hence, by the well-covering property of f, x lies within a constant number of cells away from the cell c. Therefore each candidate event of type (v) is detected in at most $O(1)$ cells, and, since in each such event at least one wave that encodes paths that enter c through e_B gets eliminated, the total number of these candidate events during the whole algorithm is bounded by the total number of generated waves, which is $O(n)$ by the proof of Lemma 5.12. □

We summarize the main result of the preceding discussion in the following lemma.

Lemma 5.14. *The total number of candidate events processed during the wavefront propagation is $O(n)$. The wavefront propagation phase of the algorithm takes a total of $O(n \log n)$ time and space.*

5.4 Shortest path queries

In this subsection we describe the second phase of the algorithm, namely, the preprocessing needed for answering shortest path queries.

Preprocessing building blocks. At the end of the propagation phase, the one-sided wavefronts for all transparent edges have been computed. Furthermore, for each building block B of a surface cell c and a topologically constrained wavefront W that was propagated in c through B, all bisector events that are true with respect to the propagation of W in B have been exactly computed. We call a generator of W *active in* B if it was detected by the algorithm to be involved in such an event inside B. Note that if W has been split in another preceding building block of c into two sub-wavefronts W_1, W_2 that now traverse B as two distinct topologically constrained wavefronts, no interaction between W_1 and W_2 in B is detected or processed (the two traversals are processed at two distinct nodes of a block tree, or different block trees of $\mathcal{T}(e)$, both representing B). Moreover, if W has been split in B (which might happen if B is a nonconvex type I block — see Section 3.1), the split portions cannot collide with each other inside B; see Figure 56. The wavefront propagation algorithm lets us compute the active generators for all pairs (W, B) in a total of $O(n \log n)$ time.

We next define the partition *local*(W, B) of the unfolded portion of a building block B that was covered by a wavefront W (and the wavefronts that W has been split into during its propagation within B), which will be further preprocessed for point location for shortest path queries. The partition *local*(W, B) consists of *active* and *inactive regions*, defined as

follows. The active regions are those portions of B that are claimed by generators of W that are active in B, and each inactive region is claimed by a contiguous band of waves of W that cross B in an "uneventful" manner, delimited by a sequence of pairwise disjoint bisectors. See Figure 56 for an illustration. Note that the complexity of $local(W, B)$ is $O(k+1)$, where k is the number of true critical events of W in B.

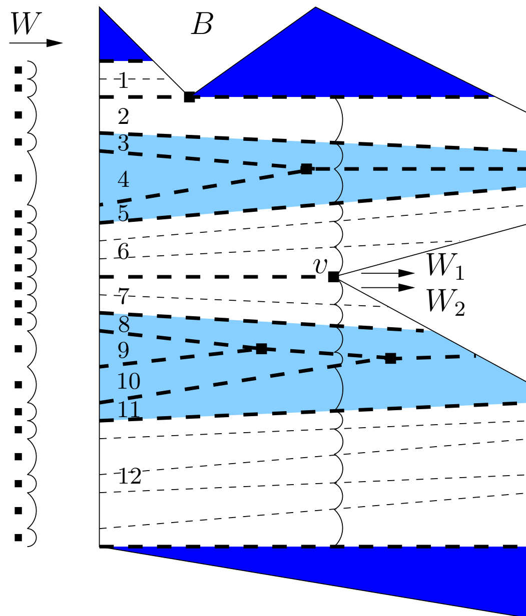

Figure 56: *The wavefront W enters the building block B (in this example, B is a nonconvex block of type I, bounded by solid lines) from the left. The partition $local(W, B)$ is drawn by thick dashed lines; thin dashed lines denote bisectors of W that lie fully in the interior of the inactive regions. The regions of the partition are numbered from 1 to 12; the active regions are lightly shaded, the inactive regions are white, and the portions of B that were not traversed by W due to visibility constraints are darkly shaded. The locations of the bisector events of W and the reflex vertices reached by W in B are marked. W is split at v into W_1 and W_2, and $local(W, B)$ includes these sub-wavefronts too.*

Here are several comments concerning this definition. The edges of $local(W, B)$ are those bisectors of pairs of generators of W, at least one of which is active in B. The first and the last bisectors of W are also defined to be edges of $local(W, B)$. If, during the propagation in B, W has been split (into sub-wavefronts W_1, W_2) at a reflex vertex v of B, then the ray from the generator of W, whose wave has been split at v, through v (an artificial extreme bisector of both W_1, W_2) is also defined to be an edge of $local(W, B)$; this ray terminates at v in one of the wavefronts W_1, W_2, and may extend beyond v in the other. If W has been split into sub-wavefronts W_1, W_2 in such a way, we treat also the bisectors of W_1, W_2 as if they belonged to W (that is, embed $local(W_1, B)$, $local(W_2, B)$ as extensions of $local(W, B)$, and preprocess them together as a single partition of B). The partition can actually be computed "on the fly" during the propagation of W in B, in additional time proportional to the number of detected critical events of W in B.

We preprocess each such partition $local(W, B)$ for point location [15, 25], so that, given a query point $p \in B$, we can determine which region r of $local(W, B)$ contains the unfolded

image q of p (that is, if B is of type II or III and $\mathcal{E}$ is the edge sequence associated with B, $q = U_{\mathcal{E}}(p)$; if B is of type I or IV then $q = p$). If r is traversed by a single wave of W (which is always the case when r is active, and can also occur when r is inactive), it uniquely defines the generator of W that claims p (if we ignore other wavefronts traversing B). This step of locating r takes $O(\log k)$ time. If q is in an inactive region r of $local(W, B)$ that was traversed by more than one wave of W, then r is the union of several "strips" delimited by bisectors between waves that were propagated through B without events. We can then SEARCH for the claimer of q in W in $O(\log n)$ time (see Section 5.1).[22]

Preprocessing S_{3D}. In order to locate the surface subdivision cell that contains the query point, we also preprocess the 3D-subdivision S_{3D} for point location, as follows. First, we subdivide each perforated cube cell into six rectilinear boxes, by extending its inner horizontal faces until they reach its outer boundary, and then extending two parallel vertical inner faces until they reach the outer boundary too, in the region between the extended horizontal faces. Next, we preprocess the resulting 3-dimensional rectilinear subdivision in $O(n \log n)$ time for 3-dimensional point location, as described in [12]. The resulting data structure takes $O(n \log n)$ space, and a point location query takes $O(\log n)$ time.

Answering shortest-path queries. To answer a shortest-path query from s to a point $p \in \partial P$, we perform the following steps.

1. Query the data structure of the preprocessed S_{3D} to obtain the 3D-cell c_{3D} that contains p.
2. Query the surface unfolding data structure (defined in Section 2.4) to find the facet f of ∂P that contains p in its closure.[23]
3. Since the transparent edges are close to, but not necessarily equal to, the corresponding intersections of subfaces of S_{3D} with ∂P, p may lie either in a surface cell induced by c_{3D} or by an adjacent 3D-cell, or in a surface cell derived from the intersection of transparent edges of $O(1)$ such cells. To find the surface cell containing p, let $I(c_{3D})$ be the set of the $O(1)$ surface cells induced by c_{3D} and by its $O(1)$ neighboring 3D-cells in S_{3D} (whose closures intersect that of c_{3D}). For each cell $c \in I(c_{3D})$, check whether $p \in c$, as follows.
   (a) Using the surface unfolding data structure, find the transparent edges of ∂c that intersect f, by finding, for each transparent edge e of ∂c, the polytope edge sequence $\mathcal{E}$ that e intersects, and searching for f in the corresponding facet sequence of $\mathcal{E}$ (see Section 2.4).
   (b) Calculate the portion $c \cap f$ and determine whether p lies in that portion.

[22]Note that we could have also found the claimer by a naive binary search through the list of generators of r, which would have cost $O(\log^2 n)$. Here SEARCH can be regarded as an optimized implementation of such a binary search.

[23]Alternatively, this step can be done using a standard point location technique, if we additionally preprocess ∂P as a planar map.

4. If p is contained in more than one surface cell, then it must be incident to at least one transparent edge. If p is incident to more than one transparent edge, then p is a transparent edge endpoint and its shortest path distance has already been calculated by the propagation algorithm and can be reported immediately. If p is incident to exactly one transparent edge e, then we SEARCH for the claimer of p in each of the two one-sided wavefronts at e, and report the one with the shortest distance to p (or report both if the distances are equal).

   If p is contained in exactly one surface cell c, then perform the following step.

5. Among the $O(1)$ building blocks of c, find a block B that contains p (if p is on a contact interval, and thus belongs to more than one building block, we can choose any of these blocks). For each wavefront W that has entered B, we find the generator s_i that claims p in W, using the point location structure of $local(W, B)$ as described above, and compute the distance $d(s_i, p)$. We report the minimal distance from s to p among all claimers of p that were found at this step.

6. If the corresponding shortest path has to be reported too, we report all polytope edges that are intersected by the path from the corresponding source image to p. If needed, all the shortest paths in $\Pi(s, p)$ (in case there are several such paths) can be reported in the same manner.

Steps 1–3 take $O(\log n)$ time, using [12] and the data structure defined in Section 2.4. As argued above, it takes $O(\log n)$ time to perform Step 4 or Step 5 (note that only one of these steps is performed). This, at long last, concludes the proof of our main result (modulo the construction of the 3D-subdivision, given in the next section):

Theorem 5.15 (Main Result). *Let P be a convex polytope with n vertices. Given a source point $s \in \partial P$, we can construct an implicit representation of the shortest path map from s on ∂P in $O(n \log n)$ time and space. Using this structure, we can identify, and compute the length of, the shortest path from s to any query point $q \in \partial P$ in $O(\log n)$ time (in the real RAM model). A shortest path $\pi(s, q)$ can be computed in additional $O(k)$ time, where k is the number of straight edges in the path.*

6 Constructing the 3D-subdivision

This section presents the proof of Theorem 2.1, by describing an algorithm for constructing a conforming 3D-subdivision for a set V of n points in $\mathbb{R}^3$. This is a straightforward generalization of the construction of a similar conforming subdivision in the plane [22], without any significant changes, except for the obvious change in dimension. Readers familiar with the construction in [22] will find the presentation below very similar, but we nevertheless describe it here for the sake of completeness.

The main part of the algorithm constructs a *1-conforming* 3D-subdivision (defined below) of size $O(n)$ in $O(n \log n)$ time, and Lemma 6.1 shows how to transform this subdivision into a conforming 3D-subdivision of size $O(n)$ in $O(n)$ additional time.

A *1-conforming* 3D-subdivision is similar to a conforming 3D-subdivision, except for the well-covering property, which is replaced by the following *1-well-covering* property. Given a 1-conforming 3D-subdivision S^1_{3D}, a subface $h \in S^1_{3D}$ is said to be *1-well-covered* if the following three conditions hold (compare with Section 2):

(W1) There exists a set of cells $C(h) \subseteq S^1_{3D}$ such that h lies in the interior of their union $R(h) = \bigcup_{c \in C(h)} c$. The region $R(h)$ is called the *well-covering region* of h.

(W2) The total complexity of all the cells in $C(h)$ is $O(1)$.

(W3$_1$) If g is a subface on $\partial R(h)$, then $d_{3D}(h, g) \geq \max\{l(h), l(g)\}$.

A subface h is *strongly 1-well-covered* if the stronger condition (W3'$_1$) holds:

(W3'$_1$) For any subface g so that h and g are incident either to nonadjacent faces of distinct cells or to nonadjacent faces of the same cell of the subdivision, $d_{3D}(h, g) \geq \max\{l(h), l(g)\}$.

If every subface of a 1-conforming 3D-subdivision S^1_{3D} is strongly 1-well-covered, then S^1_{3D} is *strongly* 1-conforming.

The minimum vertex clearance property is replaced by the following (weaker) *minimum vertex 1-clearance property*:

(MVC$_1$) For any point $v \in V$ and for any subface h, $d_{3D}(v, h) \geq \frac{1}{4} l(h)$.

Lemma 6.1. *Let V be a set of n points, and let S^1_{3D} be a 1-conforming 3D-subdivision for V of size $O(n)$ that satisfies the following additional properties:*

(1) Each face of S^1_{3D} is axis-parallel.

(2) Each cell is either a whole or a perforated cube (with subdivided faces).

(3) Each point of V is contained in the interior of a whole cube cell.

(4) The minimum vertex 1-clearance property is satisfied.

We can then construct from S^1_{3D}, in time $O(n)$, a conforming 3D-subdivision S_{3D} for V with complexity $O(n)$, that satisfies the same additional properties (1–3) and the minimum vertex clearance property (MVC) instead of (MVC$_1$). If S^1_{3D} is a strongly 1-conforming 3D-subdivision, then we can construct S_{3D} to have the above properties, and also to be a strongly conforming 3D-subdivision.

Proof: Subdivide each face of S^1_{3D} into 16×16 equal-length square subfaces. Define the well-covering region of each new subface h in S_{3D} to be the same as the well-covering region in S^1_{3D} of the subface of S^1_{3D} that contains h. These operations can be performed in $O(n)$ overall time. It is easy to check that the subdivision thus defined satisfies properties (C1–C3) of Section 2.2. We provide the proof for the sake of completeness.

(C1) S_{3D} has the same set of cells as S^1_{3D}, so each cell of S_{3D} contains at most one point of V in its closure.

(C2) To show that each subface h of S_{3D} is well-covered, we check that it satisfies conditions (W1), (W2), and (W3) (or (W3') if S^1_{3D} is strongly 1-conforming). Let h_1 be the subface of S^1_{3D} that contains h. Let $C_1(h_1)$ be the set of cells of S^1_{3D} whose union $R_1(h_1)$ is the well-covering region of h_1. Define $C(h)$ and $R(h)$ analogously.

- (W1) By definition, $R(h) = R_1(h_1)$, so h is contained in its interior.
- (W2) Each subface of each cell in $C_1(h_1)$ is divided into 16×16 pieces in $C(h)$, so the total complexity of $C(h)$ is $O(1)$.
- (W3) Let g be a subface of S_{3D} on $\partial R(h)$, and let g_1 be the subface of S^1_{3D} from which it is derived. Then we have $d_{3D}(h, g) \geq d_{3D}(h_1, g_1) \geq \max\{l(h_1), l(g_1)\} = 16 \cdot \max\{l(h), l(g)\}$.
- (W3') Similar to the argument just given for (W3).

(C3) The well-covering regions in S_{3D} are the same as in S^1_{3D}, so each contains at most one vertex of V.

The properties (1–3) are satisfied in S_{3D} since they are satisfied in S^1_{3D}. The minimum vertex clearance property is satisfied in S_{3D} since the minimum vertex 1-clearance property is satisfied in S^1_{3D}, and since in S^1_{3D} the edge of a subface is 16 times longer than the edges of the subfaces derived from it in S_{3D}, while the distance between two subfaces in S_{3D} is not shorter than the distance between the original subfaces in S^1_{3D} (similarly, the distance between a subface in S_{3D} and a point $v \in V$ is not shorter than the distance between v and the original subface in S^1_{3D}). This establishes the lemma. □

6.1 i-Boxes and i-quads

Before we describe the construction of the 1-conforming 3D-subdivision, we need a few definitions.

We fix a Cartesian coordinate system in $\mathbb{R}^3$. For any whole number i, the ith-order grid $\mathcal{G}_i$ in this system is the arrangement of all planes $x = k2^i, y = k2^i$ and $z = k2^i$, for $k \in \mathbb{Z}$. Each cell of $\mathcal{G}_i$ is a cube of size $2^i \times 2^i \times 2^i$, whose near-lower-left[24] corner lies at a point $(k2^i, l2^i, m2^i)$, for a triple of integers k, l, m. We call each such cell an *i-box*.

Any $4 \times 4 \times 4$ contiguous array of i-boxes is called an *i-quad*. Although an i-quad has the same size as an $(i+2)$-box, it is not necessarily an $(i+2)$-box because it need not be a cell in $\mathcal{G}_{i+2}$; see Figure 57 for the planar analog of i-boxes and i-quads. The eight non-boundary i-boxes of an i-quad form its *core*, which is thus a $2 \times 2 \times 2$ array of i-boxes. Observe that an i-box b has exactly eight i-quads that contain b in their cores.

The algorithm constructs a conforming partition of the point set V in a bottom-up fashion. It simulates a growth process of a cube box around each data point, until their

[24]This means near in the y-direction, lower in the z-direction, and left in the x-direction.

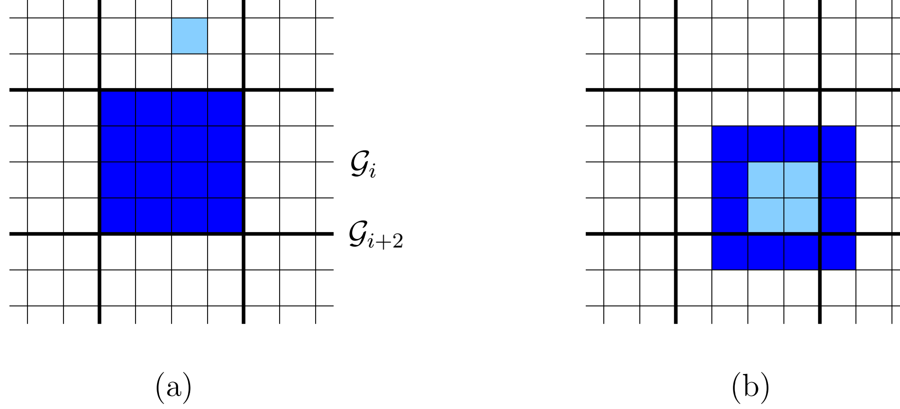

Figure 57: *The planar analog of (a) an i-box (lightly shaded) and an $(i+2)$-box (darkly shaded); and (b) an i-quad (darkly shaded) and its core (lightly shaded).*

union becomes connected. The simulation works in discrete *stages*, numbered $-2, 0, 2, 4, \ldots$. It produces a subdivision of space into axis-parallel cells. The key object associated with a data point p at stage i is an i-quad containing p in its core. In fact, the following stronger condition holds inductively: Each $(i-2)$-quad constructed at stage $(i-2)$ lies in the core of some i-quad constructed at stage i.

In each stage, we maintain only a minimal set of quads. The set of i-quads maintained at stage i is denoted as $\mathcal{Q}(i)$. This set is partitioned into equivalence classes under the transitive closure of the *overlap* relation, where two i-quads overlap if they have a common i-box (not necessarily in their cores). That is, regarding the quads as open, two quads q, q' are equivalent if and only if they belong to the same connected component of the union of the current quads. Let $S_1(i), \ldots, S_k(i)$ denote the partition of $\mathcal{Q}(i)$ into equivalence classes, and let $\equiv_i$ denote the equivalence relation.

The portion of space covered by quads in one class of this partition is called a *component*. Each component at stage i is either an i-quad or a connected union of (open) i-quads. We classify each component as being either *simple* or *complex*. A component at stage i is *simple* if (1) its outer boundary is an i-quad and (2) it contains exactly one $(i-2)$-quad of $\mathcal{Q}(i-2)$ in its core. Otherwise, the component is *complex*.

6.2 The invariants

As the algorithm progresses, we construct the faces that constitute the boundaries of certain components, where each boundary face is an axis-parallel square. Together, these faces subdivide $\mathbb{R}^3$ into axis-parallel cells, which comprise the subdivision. The critical property of the subdivision is the following *conforming property*:

> (CP) For any two subfaces h, g that are incident to either nonadjacent faces of distinct cells or to nonadjacent faces of the same cell of the subdivision, $d_{3D}(h, g) \geq \max\{l(h), l(g)\}$.

The algorithm constructs subfaces whose edge lengths keep increasing, and we never subdivide previously constructed subfaces. In order to help maintain (CP), we will also enforce the following auxiliary property.

> ($\mathrm{CP}_{\mathrm{aux}}$) The boundary of each complex component at stage i is subdivided into square subfaces that are faces of the ith-order grid $\mathcal{G}_i$.

The algorithm delays the explicit construction of the outer boundary of a simple component until just before it merges with another component to form a complex component. This is crucial to ensure that the final subdivision has only $O(n)$ size, and can be constructed in near-linear time.

The algorithm consists of two main parts. The first part grows the $(i-2)$-quads of stage $(i-2)$ into i-quads of stage i, and the other part computes and updates the equivalence classes, and constructs subdivision subfaces that satisfy properties ($\mathrm{CP}_{\mathrm{aux}}$) and (CP). These tasks are performed by the procedures *Growth* and *Build-subdivision*, respectively. We postpone the discussion of *Growth* to a later subsection, but introduce the necessary terminology to allow us to describe *Build-subdivision*.

Given an i-quad q, $Growth(q)$ is an $(i+2)$-quad containing q in its core. For a *family* S of i-quads, $Growth(S)$ is a *minimal* (but not necessarily the minimum) set of $(i+2)$-quads such that each i-quad in S is contained in a member of $Growth(S)$.

As mentioned earlier, up to eight $(i+2)$-quads may qualify for the role of $Growth(q)$, for an i-quad q, but for now we let $Growth(q)$, or $\tilde{q}$, denote the unique $(i+2)$-quad returned by the procedure *Growth* (see below for details concerning its choice).

6.3 The *Build-subdivision* procedure

By appropriate scaling and translation of 3-space, we may assume that the L_∞-distance between each pair of points in V is at least 1, and that no point coordinate is a multiple of $\frac{1}{16}$. For each point $p \in V$, we *construct* (to distinguish from other quads that we only *compute* during the process, constructing a quad means actually adding it to the 3D-subdivision) a (-4)-quad with p at the near-lower-left (-4)-box of its core; this choice ensures that the minimal distance from p to the boundary of its quad is at least $\frac{1}{4}$ of the side length of the quad. Around each of these quads q, we compute (but *not* construct yet) a (-2)-quad with q in its core, so that when there is more than one choice to do that (there are one, two, four, or eight possibilities to choose the (-2)-quad if ∂q is coplanar with none, two, four, or six planes of $\mathcal{G}_{-2}$, respectively), we always choose the (-2)-quad whose position is extreme in the near-lower-left direction. This ensures that the (-2)-quads associated with distinct points are openly disjoint (because the points of V are at least 1 apart from each other in the L_∞-distance; without the last constraint, one could have chosen two (-2)-quads whose interiors have nonempty intersection).

These quads form the set $\mathcal{Q}(-2)$, which is the initial set of quads in the *Build-subdivision* algorithm described below. Each quad in $\mathcal{Q}(-2)$ forms its own singleton component under the equivalence class in stage -2. As above, we regard all quads in $\mathcal{Q}(-2)$ as open, and thus

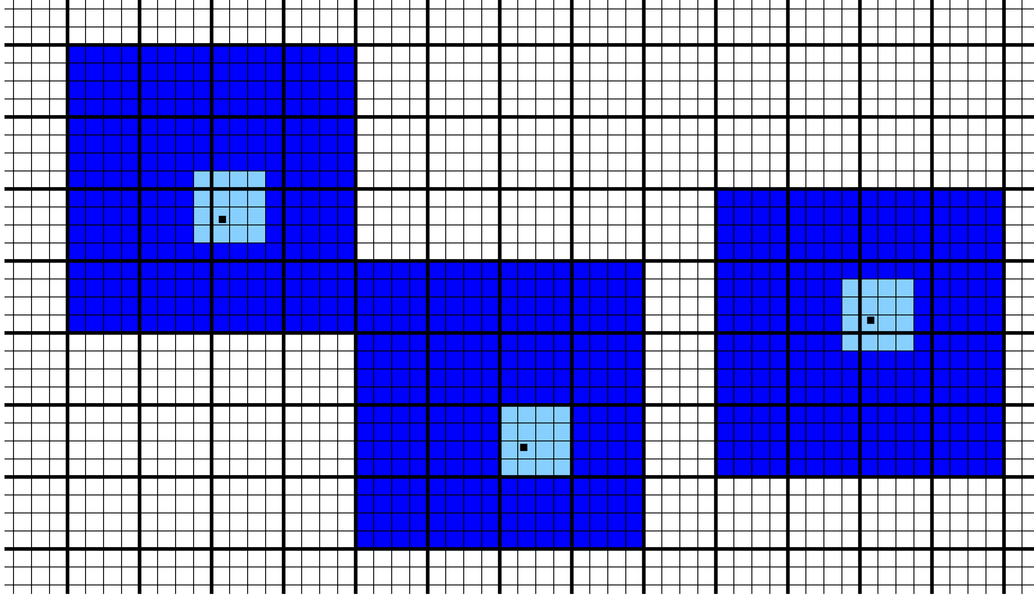

Figure 58: *Planar analog of three (not necessarily constructed yet) initial components of* (-2)-*quads* ($\mathcal{G}_{-2}$ *is drawn with thick lines;* $\mathcal{G}_{-4}$ *is also shown). An initial* (-4)-*quad (lightly shaded) around each point* $p \in V$ *(in its core) is contained in the core of the corresponding* (-2)-*quad (darkly shaded). Each of the (open)* (-2)-*quads is an initial simple component. They are all pushed as far as possible in the lower-left direction.*

forming distinct simple components, even though some pairs might share boundary points. See Figure 58 for an illustration of the planar analog of this initial structure. Both properties ($\mathrm{CP}_{\mathrm{aux}}$) and (CP) are clearly satisfied at this stage.

The pseudo-code below describes the details of the algorithm *Build-subdivision*. This pseudo-code is not particularly efficient; an efficient implementation is presented later in Section 6.5.

Procedure *Build-subdivision*

Initialize $\mathcal{Q}(-2)$ (* *as described above.* *)

$i := -2$.

while $|\mathcal{Q}(i)| > 1$ **do**

1. $i := i + 2$.
2. (* *Compute* $\mathcal{Q}(i)$ *from* $\mathcal{Q}(i-2)$. *)
   (a) Initialize $\mathcal{Q}(i) := \emptyset$.
   (b) **for** each equivalence class S of $\mathcal{Q}(i-2)$ **do**
       $\mathcal{Q}(i) := \mathcal{Q}(i) \cup Growth(S)$.
   (c) **for** each pair of i-quads $q, q' \in \mathcal{Q}(i)$ **do**
       if $q \cap q' \neq \emptyset$ set $q \equiv_i q'$.
   (d) Extend $\equiv_i$ to an equivalence relation by transitive closure, and compute the resulting equivalence classes.
3. (* *Process simple components of* $\equiv_{i-2}$ *that are about to merge with some other component.* *)
   for each $q \in \mathcal{Q}(i-2)$ **do**
   (a) Let $\tilde{q} := Growth(q)$ as computed in Step 2b.
   (b) **if** q forms a single simple component at stage $(i-2)$
       but $\tilde{q}$ does not form a single simple component at stage i **then**
       Construct the boundary of q and subdivide each of its faces
       into 4×4 subfaces by the planes of $\mathcal{G}_{i-2}$.
4. (* *Process complex components.* *)
   for each equivalence class S of $\mathcal{Q}(i)$ **do**

   Let $S' := \{q \in \mathcal{Q}(i-2) \mid Growth(q) \in S\}$.

   if $|S'| > 1$ **then** (* *S is complex; see Figure 59 for a planar analog.* *)
   (a) Let $R_1 := \bigcup_{q \in S'} \{\text{the core of } Growth(q)\}$.
   (b) Let $R_2 := \bigcup_{q \in S'} q$.
   (c) **Construct** $(i-2)$-boxes to fill $R_1 \setminus R_2$.
   (d) **Construct** i-boxes to fill $S \setminus R_1$; partition each cell boundary with an endpoint incident to R_1 into 4×4 subfaces of side length 2^{i-2}, to satisfy properties ($\mathrm{CP}_{\mathrm{aux}}$) and (CP).

endwhile

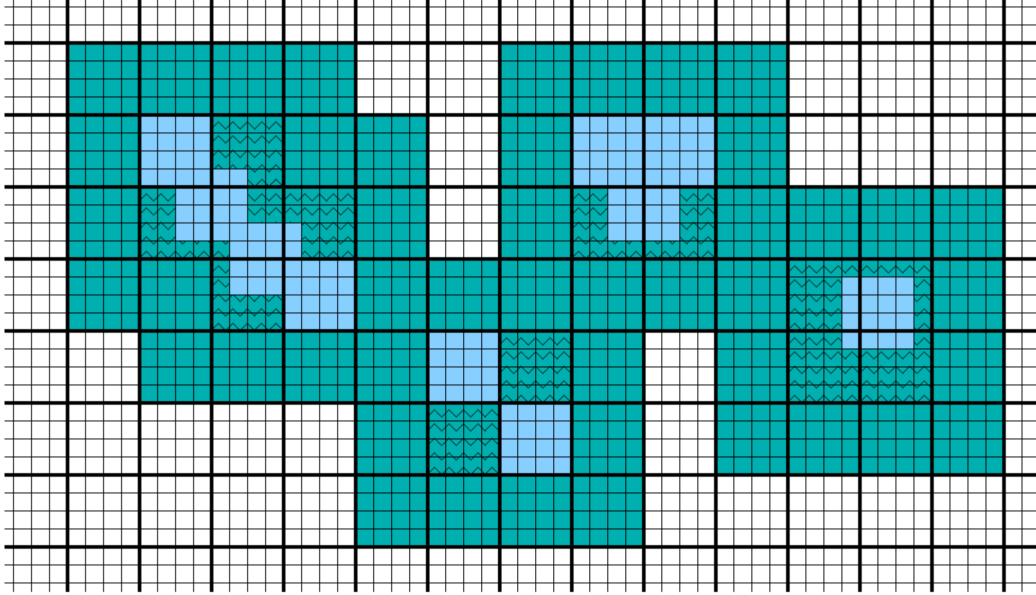

Figure 59: *A planar analog of a complex component $S \subseteq \mathcal{Q}(i)$ (darkly shaded), consisting of five i-quads (the grid $\mathcal{G}_i$ is drawn with thick lines; $\mathcal{G}_{i-2}$ is also shown). The $(i-2)$-quads of $\mathcal{Q}(i-2)$ in the cores of these i-quads (lightly shaded) constitute R_2. The portion $R_1 \setminus R_2$ (where R_1 is the union of the cores of the i-quads of S) is drawn with zigzag pattern.*

Lemma 6.2. *The subdivision computed by the algorithm* Build-subdivision *satisfies properties* ($\mathrm{CP}_{\mathrm{aux}}$) *and* (CP)*.*

Proof: We prove by induction that the properties hold for each of the families of quads $\mathcal{Q}(i)$, for all i. The initial family of quads $\mathcal{Q}(-2)$ clearly satisfies the two properties. We argue that no step of the algorithm *Build-subdivision* ever violates these properties.

Step 2 computes $Growth(S)$ for each equivalence class of $\mathcal{Q}(i-2)$ and then computes $\mathcal{Q}(i)$. No new subfaces are constructed in this step.

The only subfaces constructed in Step 3 are on the boundaries of simple components. Let q be an $(i-2)$-quad that is a simple component of $\mathcal{Q}(i-2)$. By definition, the single $(i-4)$-quad of $\mathcal{Q}(i-4)$ contained in q lies in its core and thus its L_∞-distance from the outer boundary of q is at least 2^{i-2}. Hence the subfaces already constructed in the core satisfy the property (CP) — they have length no more than 2^{i-2} (actually 2^{i-4}, except when $i = 0$), and are separated from the boundary of q by a distance of at least 2^{i-2}. We construct the boundary of q in Step 3; since any previously constructed subfaces within q lie in its core, the new subfaces satisfy (CP) with respect to these previously constructed subfaces. The new subfaces on ∂q satisfy (CP) also with respect to the new subfaces on the boundary of any other $(i-2)$-quad, since all $(i-2)$-quads are part of $\mathcal{G}_{i-2}$ and therefore either their boundaries intersect or there is a gap of at least 2^{i-2} between them. ($\mathrm{CP}_{\mathrm{aux}}$) holds vacuously, since it involves only complex components (that are not processed in this step).

Step 4 subdivides each complex component $S \subseteq \mathcal{Q}(i)$. Again, the distance between the boundary of S and any component of $\mathcal{Q}(i-2)$ that it contains is at least the width of an

i-box. By the property ($\mathrm{CP}_{\mathrm{aux}}$) (at stage $i-2$), the $(i-2)$-boxes constructed at Step 4c satisfy (CP) with respect to the previously constructed subfaces; they clearly satisfy (CP) with respect to the subfaces of other $(i-2)$-boxes constructed in this stage. Step 4d packs the area between the core and the boundary of S with i-boxes, and breaks the newly-constructed faces that are incident to previously constructed cells into 4×4 subfaces, to guarantee (CP) with respect to those cells. (The previously constructed subfaces on the core boundary have length 2^{i-2}, so, by induction, the cells incident to them have side lengths at least 2^{i-2}. It follows that the cells inside the core satisfy (CP) with respect to the newly constructed subfaces of length 2^{i-2}.) The subfaces on the boundary of S (of length 2^i) are unbroken, so ($\mathrm{CP}_{\mathrm{aux}}$) holds at the end of stage i. These subfaces also satisfy (CP) with respect to subfaces on the boundary of any other i-quad, since all i-quads are part of $\mathcal{G}_i$. This completes the proof. □

Lemma 6.3. *The subdivision produced by* Build-subdivision *has size* $O(n)$.

Proof: We show that the algorithm constructs a linear number of subfaces altogether. The number of subfaces constructed in Step 3 is proportional to the number constructed in Step 4 — we construct a constant number of subfaces in Step 3 for each simple component that merges to form a complex component at the next stage. The number of subfaces constructed in Step 4 for a complex component S is $O(|S'|)$ (where $S' = \{q \in \mathcal{Q}(i-2) \mid Growth(q) \in S\}$), the number of $(i-2)$-quads whose growths constitute S. The key observation in proving the linear bound is that the total size of $\mathcal{Q}$ decreases every two stages by an amount proportional to the total number of quads in complex components. This fact, which we prove in the next subsection (Lemma 6.5), can be expressed as follows. If f_i subfaces are constructed at stage i, then

$$|\mathcal{Q}(i+2)| \leq |\mathcal{Q}(i-2)| - \beta f_i,$$

for some absolute constant β. That is,

$$\beta f_i \leq |\mathcal{Q}(i-2)| - |\mathcal{Q}(i+2)|.$$

If we sum this inequality over all even $i \geq 0$, the right-hand side telescopes, and we obtain

$$\beta \textstyle\sum_i f_i \leq |\mathcal{Q}(-2)| + |\mathcal{Q}(0)|.$$

Since $|\mathcal{Q}(-2)| = n$, we have $|\mathcal{Q}(0)| \leq n$ and therefore $\sum_i f_i \leq \frac{2n}{\beta} = O(n)$, as asserted. □

Lemma 6.4. *The subdivision* S^1_{3D} *that* Build-subdivision *produces is strongly 1-conforming and satisfies the following additional properties:*

(1) Each face of S^1_{3D} *is an axis-parallel square.*

(2) Each cell is either a whole or a perforated cube (with subdivided faces).

(3) Each input point is contained in the interior of a whole cube cell.

(4) The minimum vertex 1-clearance property is satisfied.

Proof: Strong 1-conformity is a consequence of (CP), as we now show. Condition (C1) is trivially true, since each point is initially enclosed by a cube. To establish the 1-well-covering property (Condition (C2)), let $I(h)$ be the union of the $O(1)$ cells incident to a subface h. $I(h)$ contains at most eight cells. Indeed, a subface h cannot be incident to a cell c whose side length is less than $4l(h)$, by construction, and h can be incident to at most eight cells of side length greater than or equal to $4l(h)$; see Figure 60 for an illustration. By (CP), the distance from h to any subface outside or on the boundary of $I(h)$ is at least $l(h)$. The subface h may be coplanar with other subfaces of the two cells on whose boundary it lies. We define $C(h)$ to be the set of cells incident to one of these coplanar subfaces; $R(h)$, the union of these cells, is a superset of $I(h)$. See Figure 61 for an illustration.

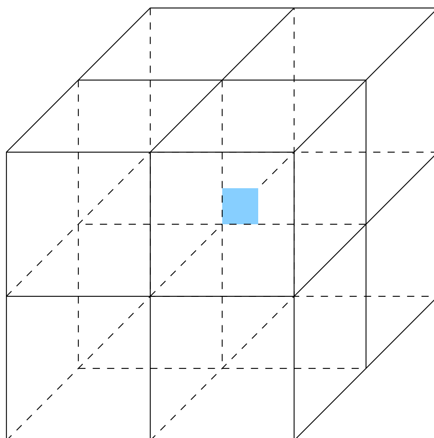

Figure 60: *A subface h (shaded) can be incident to at most eight 3D-cells of side length greater than or equal to $4l(h)$.*

Since the two cells with h as a boundary subface meet only along subfaces coplanar with h, the definition of $R(h)$ implies that for any subface g on or outside the boundary of $R(h)$, $I(g)$ does not contain both cells incident to h. But this implies, by (CP), that h is on or outside the boundary of $I(g)$, and hence the distance from h to g is at least $l(g)$. The subface h certainly lies in the interior of $R(h)$ (Condition (W1)). Condition (W2) follows because $C(h)$ is the union of $I(h')$ for $O(1)$ subfaces h' coplanar with h, $|I(h')| \leq 8$ for each h', and each cell has a constant number of faces. As noted above, the minimum distance between h and any subface g on or outside the boundary of $R(h)$ is at least $\max\{l(h), l(g)\}$, which establishes Condition ($W3_1$). (The stronger condition ($W3'_1$) is the property (CP) itself.) Condition (C3) follows from the observation that a well-covering region $R(h)$ contains a vertex v of V if and only if h is a subface of the cube containing v. This is because each vertex-containing cube is the inner cube of a perforated cube in the subdivision. No subface belongs to two such cubes, so Condition (C3) holds.

To establish the minimum vertex 1-clearance property, consider a subface h of the (whole) cube cell c_{3D} that closes a point $v \in V$. By construction, $d_{3D}(v, h) \geq \frac{1}{16}$ and $l(h) = \frac{1}{4}$, so the property holds for h. For any other subface g of the 3D-subdivision, the shortest straight

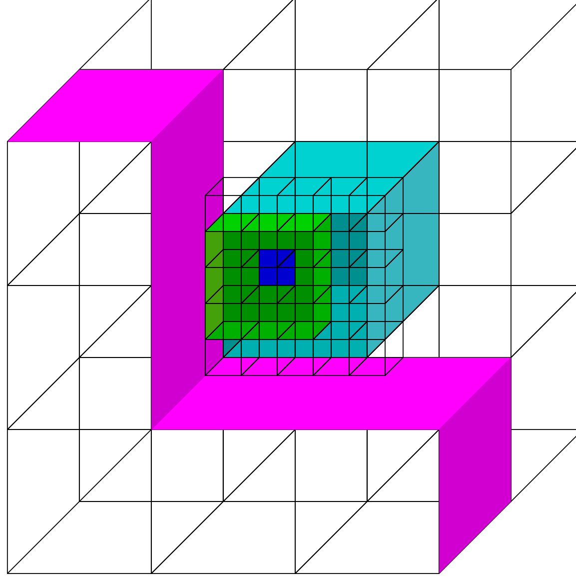

Figure 61: *The region $I(h)$ of the darkly shaded face h contains, in this example, a total of ten 3D-cells (3×3 shaded smaller cells and one shaded larger cell behind them). The well-covering region $R(h)$ contains all the 39 cells drawn in this figure: 5×5 smaller cells, five larger cells on the front, covered on top by a darkly shaded "carpet", and 3×3 larger cells on the back.*

line from g to v goes through some h that lies in ∂c_{3D}. By (CP), $d_{3D}(g, h) \geq l(g)$, and since $d_{3D}(v, g) \geq d_{3D}(g, h)$, the minimum vertex 1-clearance property holds for g.

The remaining properties (1–4) hold by construction. This completes the proof. $\square$

6.4 The *Growth* procedure

In this subsection we describe the algorithm for computing $Growth(S)$ for a set of i-quads S, and show that the number of quads decreases every two stages (that is, from stage i to stage $i+4$) by an amount proportional to the total complexity of the complex components. Let $S \subset \mathcal{Q}(i)$ be a set of i-quads forming a complex component under the equivalence relation $\equiv_i$. Recall that we want $Growth(S)$ to be minimal (albeit not necessarily the minimum) set of $(i+2)$-quads such that each i-quad of S lies in the core of some $(i+2)$-quad in $Growth(S)$. We will show that

$$|Growth(Growth(S))| \leq \kappa\, |S|,$$

for an absolute constant $0 < \kappa < 1$. The pseudo-code below describes an unoptimized version of the algorithm for computing $Growth(S)$. The algorithm works by building a graph on the quads in S; we denote the set of the graph edges by E.

PROCEDURE *Growth*

0. Set $Growth(S) := \emptyset$ and $E := \emptyset$.

1. **foreach** pair of quads $q_1, q_2 \in S$ **do**
   if $q_1 \cup q_2$ can be enclosed in a $2 \times 2 \times 2$ array of $(i+2)$-boxes, **then**
   Add (q_1, q_2) to E.

2. Compute a maximal (not necessarily the maximum) matching M in the graph computed in Step 1, by traversing all edges of E.

3. **foreach** edge (q_1, q_2) in M **do**
   Choose an $(i+2)$-quad $\tilde{q}$ containing q_1, q_2 in its core.
   Set $Growth(q_1) := Growth(q_2) := \tilde{q}$, and add $\tilde{q}$ to $Growth(S)$.

4. **foreach** unmatched quad $q \in S$ **do**
   Set $Growth(q) := \tilde{q}$, where $\tilde{q}$ is any $(i+2)$-quad containing q in its core.
   Add $\tilde{q}$ to $Growth(S)$.

The maximum node degree of the graph constructed in Step 1 is $O(1)$ since only a constant number of i-quads can touch or intersect any i-quad q. Thus, a maximal matching in this graph has $\Theta(|E|)$ edges. Each i-quad at stage i maps to an $(i+2)$-quad at stage $(i+2)$. Since each matching edge corresponds to two i-quads that map to the same $(i+2)$-quad, it clearly follows that

$$|Growth(S)| = |S| - |M| = |S| - \Theta(|E|).$$

The crucial property is that $|E|$ is a constant fraction of $|S|$ at the next stage (that is, at stage $(i+2)$).

Lemma 6.5. *Let $S \subset \mathcal{Q}(i)$ be a set of two or more i-quads such that $Growth(S)$ is a complex component under the equivalence relation $\equiv_{i+2}$. Then $|Growth(Growth(S))| \leq \kappa\,|S|$, for an absolute constant $0 < \kappa < 1$.*

Proof: We show that either $|Growth(S)| < \frac{3}{4}\,|S|$, or at least half of the quads of $Growth(S)$ are non-isolated in the $(i+2)$-graph; that is, can be enclosed in a $2\times2\times2$ array of $(i+2)$-boxes with some other quad of $Growth(S)$.

If $|Growth(S)| < \frac{3}{4}\,|S|$, then we are done, because then we have $|Growth(Growth(S))| \leq |Growth(S)| \leq \frac{3}{4}\,|S|$. Therefore, suppose that $|Growth(S)| \geq \frac{3}{4}\,|S|$. Then at least half the i-quads of S are not matched in Step 2 of stage i of the construction of *Growth* (since there are at most $\frac{1}{4}\,|S|$ matched pairs), and their growths, which are all distinct, by construction, contribute more than half of the $(i+2)$-quads of $Growth(S)$. Consider one such unmatched i-quad $q \in S$. Since S is a non-singleton equivalence class (if it were a singleton, $Growth(S)$ would not have been complex), there exists another i-quad $q' \in S$ that overlaps q. Let $\tilde{q} = Growth(q)$ and $\tilde{q}' = Growth(q')$. By assumption, $\tilde{q} \neq \tilde{q}'$. The cores of $\tilde{q}$ and $\tilde{q}'$ both

contain the overlap region $q \cap q'$, so the cores must overlap. Therefore both cores are contained within a $3 \times 3 \times 3$ array of $(i+2)$-boxes, and both the $(i+2)$-quads $\tilde{q}$ and $\tilde{q}'$ are contained within a $5 \times 5 \times 5$ array of $(i+2)$-boxes. This ensures that $\tilde{q}$ and $\tilde{q}'$ are joined by an edge in the graph of $Growth(S)$: Any two $(i+2)$-quads that are both contained in a $5 \times 5 \times 5$ array of $(i+2)$-boxes can be covered by a $2 \times 2 \times 2$ array of $(i+4)$-boxes. See Figure 62 for an illustration.

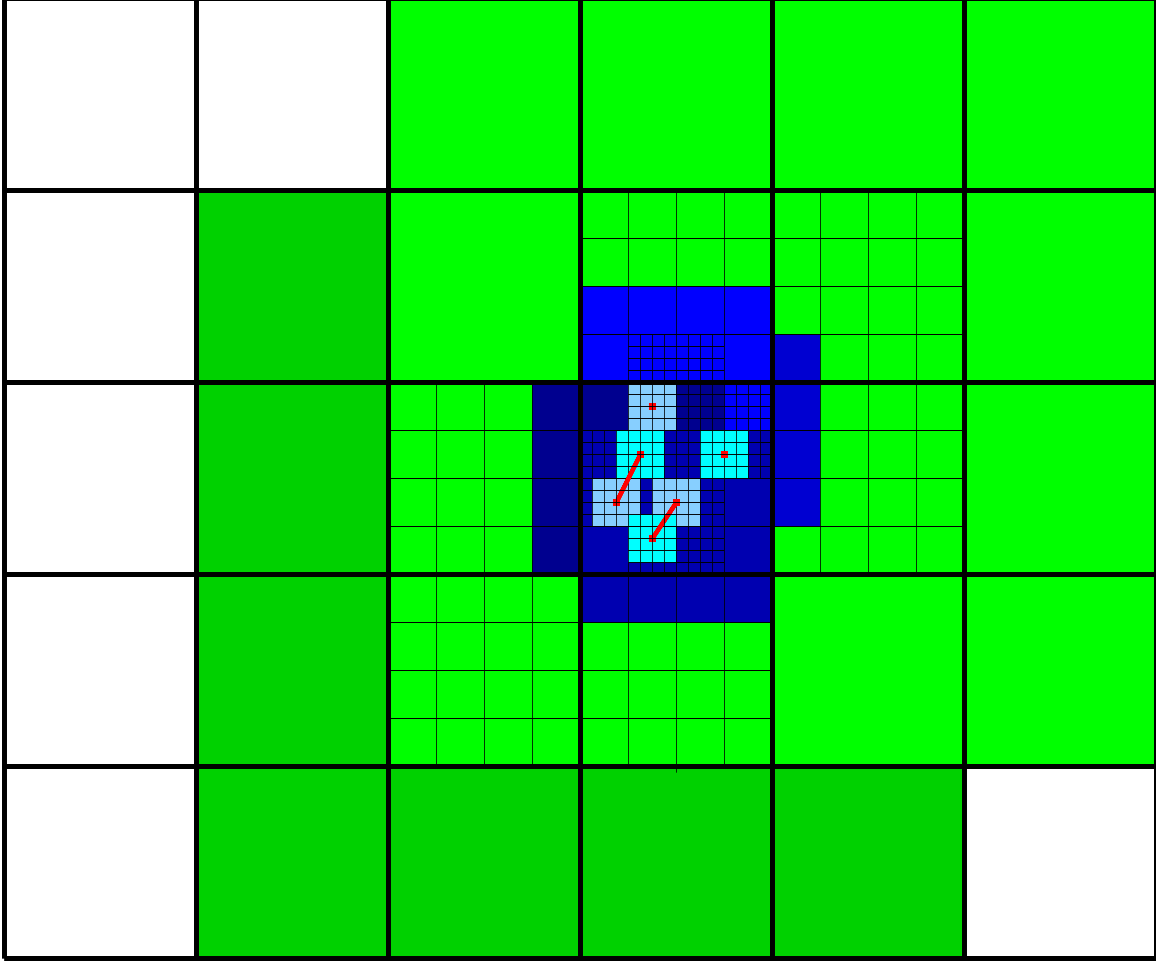

Figure 62: *The component S contains six i-quads (lightly shaded small squares). The matching (which is maximal but not the maximum) is illustrated by edges connecting the centers of the corresponding i-quads. Four (darkly shaded) $(i+2)$-quads computed by $Growth(S)$, that contain the six i-quads in their cores, overlap, so that they are contained in the cores of two (lightly shaded) $(i+4)$-quads computed by $Growth(Growth(S))$. (Grid lines are shown only where they are relevant.)*

That is, in the graph of $Growth(S)$, $|E| \geq \frac{1}{4}|S|$ (because, for each unmatched quad $q \in S$, the corresponding $(i+2)$-quad $Growth(q)$ is incident to some edge of the new graph), and since the degree of each vertex of the graph is constant, the number of edges in the maximal matching of $Growth(S)$ is $\Omega(|S|)$. This proves the inequality $|Growth(Growth(S))| \leq \kappa |S|$ for some $\kappa < 1$. $\square$

Since the number f_i of subfaces constructed at stage i is proportional to the number of $(i-2)$-quads whose growths belong to complex components, the preceding lemma establishes the earlier claim that

$$|\mathcal{Q}(i+2)| \leq |\mathcal{Q}(i-2)| - \beta f_i,$$

for some absolute constant β. For any $q, q' \in S$, we have $Growth(q) = Growth(q')$ only if q and q' touch or intersect each other, that is, their closures intersect, for otherwise q and q' cannot be contained in the core of the same $(i+2)$-quad. We use this fact to implement the procedure $Growth(S)$, for a complex component S at a fixed stage i, to run in time $O(|S| \log |S|)$: Each quad of S touches at most a constant number of other quads, and we can compute which pairs of quads touch each other, in $O(|S| \log |S|)$ time and linear space, using the algorithm of Callahan and Kosaraju [9] that finds k nearest neighbors of each point among n points in $\mathbb{R}^3$, in $O(n \log n + kn)$ time: Since there are only $k = 7^3 - 1$ distinct i-quads that can touch or intersect a given i-quad q, and these k i-quads must be L_∞-closer to q than any other i-quad, we can compute the k nearest neighbors of q, and then to check which of them touches or intersects q, in constant time for each q. From the set of touching pairs we can compute the graph edges in Step 1 of $Growth(S)$ in $O(|S|)$ additional time. All other steps of $Growth(S)$ take time proportional to the graph size, which is $O(|S|)$.

6.5 An efficient implementation of *Build-subdivision*

In order to keep the time complexity of *Build-subdivision* independent of the distribution of the points, we process a simple component only when it is about to merge with another component. This makes the processing time proportional to the number of boundary subfaces *constructed* at any stage. Except for Step 2(b), which implements *Growth*, and Step 2(c), which detects overlapping i-quads, all other steps can be implemented to run in time proportional to the number of subfaces constructed in the subdivision. (Steps 3 and 4 use the adjacency information computed in Step 2(c) to run in linear time.) In what follows we show how to use a minimum spanning tree construction to implement Steps 2(b) and 2(c) in $O(n \log n)$ time.

6.5.1 The merging of i-quads

Before we present the algorithm, we discuss the distance properties satisfied by points that lie in the same equivalence class in stage i. We say that a quad q is a *containing i-quad* of a point $u \in V$ if $q \in \mathcal{Q}(i)$ and u lies in the core of q. A point u *belongs* to an equivalence class $S \subseteq \mathcal{Q}(i)$ if there is a containing i-quad of u in S.

Lemma 6.6. *Let $u \in V$ and let $q \in \mathcal{Q}(i)$ be a containing i-quad of u. Then the L_∞-distance between u and any point on the outer boundary of q is between 2^i and $3 \cdot 2^i$.*

Proof: Since q has side length 2^{i+2}, and u lies at least a quarter of this distance away from the outer boundary, because it lies in the core, the lemma follows. □

Lemma 6.7. *Let u and v be two points of V that belong to different equivalence classes of $\equiv_i$. Then $d_\infty(u,v) > 2 \cdot 2^i$.*

Proof: Let q_u and q_v be two containing i-quads of u and v, respectively. Since u and v lie in different equivalence classes, these i-quads do not openly intersect (recall that this is a consequence of the special alignment of the i-quads). By Lemma 6.6, each of the points lies at distance at least 2^i away from the outer boundaries of their i-quads, from which the lemma follows immediately. □

Lemma 6.8. *Let $u, v \in V$ and let q_u, q_v, respectively, be the i-quads of $\mathcal{Q}(i)$ containing them. If $q_u \cap q_v \neq \emptyset$, then $d_\infty(u,v) < 6 \cdot 2^i$.*

Proof: By Lemma 6.6, the maximum L_∞ distance between u and any point on the outer boundary of q_u is at most $3 \cdot 2^i$. The same holds for v and q_v, which implies the asserted upper bound on $d_\infty(u,v)$. □

6.5.2 An efficient implementation based on L_∞-minimum spanning trees

Let V_S be the set of points of V in the cores of the i-quads of a component $S \subseteq \mathcal{Q}(i)$. The implementation of *Build-subdivision* is based on the observation that the longest edge of the L_∞-minimum spanning tree of V_S has length less than $6 \cdot 2^i$. To make this observation more precise, we define $G(i)$ to be the graph on V containing exactly those edges whose L_∞ length is at most $6 \cdot 2^i$, and define $\mathrm{MSF}(i)$ to be the minimum spanning forest of $G(i)$.

Lemma 6.9. *For each component S of $\mathcal{Q}(i)$, the points of V_S belong to a single tree of* $\mathrm{MSF}(i)$.

Proof: By Lemma 6.8, the points of V_S can be linked by a tree with edges of length shorter than $6 \cdot 2^i$. For any bipartition of the points of V_S, the minimum-weight edge linking the two subsets is shorter than $6 \cdot 2^i$. Hence, all the edges of the minimum spanning tree of V_S are shorter than $6 \cdot 2^i$, and therefore V_S belongs to a single tree of $\mathrm{MSF}(i)$. □

Lemma 6.10. *If two i-quads q_1 and q_2 belong to different components of $\mathcal{Q}(i)$, then their points belong to different trees of* $\mathrm{MSF}(i-2)$.

Proof: Any segment connecting a point of V in the component of q_1 to any point of V outside that component has length greater than $2 \cdot 2^i$, by Lemma 6.7. The points of V in the quads q_1 and q_2 are in the same tree of $\mathrm{MSF}(i-2)$ only if every bipartition of V that separates the points of q_1 from those of q_2 is bridged by an edge of length less than $6 \cdot 2^{i-2}$. But the bipartition separating the points of the component of q_1 of $\mathcal{Q}(i)$ from the rest of V has bridge length greater than $2 \cdot 2^i > 6 \cdot 2^{i-2}$. □

The algorithm is based on an efficient construction of $\mathrm{MSF}(i)$ for all i such that $\mathrm{MSF}(i) \neq \mathrm{MSF}(i-2)$. We first find all the $O(n)$ edges of the final MSF of V (a single tree), using the $O(n \log n)$ algorithm of Krznaric et al. [26] for computing an L_∞-minimum spanning tree in three dimensions. Then, for each edge e constructed by the algorithm, we compute

the stage $k = 2\left\lceil \frac{1}{2}\log_2 \frac{1}{6}|e|\right\rceil$, at which e is added to MSF(k). By processing the edges in increasing length order, we obtain the entire sequence of forests MSF(i), for those i for which MSF(i) $\neq$ MSF($i-2$).

The implementation of *Build-subdivision* below replaces Steps 1 and 2 of the original *Build-subdivision* with more efficient code based on the minimum spanning tree construction. First, we process only stages at which something happens: MSF(i) changes, or there are complex components of $\mathcal{Q}(i)$ whose *Growth* computation is nontrivial. (This optimization only significant when the ratio of maximum to minimum point separation is greater than exponential in n.) Second, we compute *Growth*(S) only for complex components and for simple components that are about to be merged with another component, and maintain the equivalence classes of $\mathcal{Q}(i)$ only for this same subset of quads. Simple components that are well separated from other components are not involved in the computation at stage i.

EFFICIENT IMPLEMENTATION OF *Build-subdivision*

For each tree T in MSF(i), maintain the corresponding set $\mathcal{Q}(i,T)$ of i-quads in $\mathcal{Q}(i)$ that are the containing quads of the vertices of T.

Initialize $i := -2$. Initialize MSF(-2) to be a forest of singleton vertices. For each vertex $v \in V$, $\mathcal{Q}(-2,\{v\})$ is a singleton quad grown around a (-4)-quad with v in its core, as described above.

Maintain a set $\mathcal{N}$ of trees in MSF(i) such that for each $T \in \mathcal{N}$, $|\mathcal{Q}(i,T)| > 1$; that is, the component of T is not a singleton quad. Initialize $\mathcal{N} := \emptyset$.

```
while |Q(i)| > 1 do
    i_old := i;
    if |N| > 0 then i := i + 2;
    else Set i to the smallest (even) i' > i such that MSF(i') ≠ MSF(i).
    (* Prepare Q(i − 2, T) before it is used to compute Q(i, T). *)
    foreach edge e of MSF(i) \ MSF(i_old) do
        Let T_1 and T_2 be the trees linked by e.
        foreach T_x ∈ {T_1, T_2} do
            if T_x ∈ N then remove T_x from N;
            else Set Q(i − 2, T_x) to be the singleton (i − 2)-quad corresponding to T_x.
        end
        Join T_1 and T_2 to get T', and add T' to N.
        Set Q(i − 2, T') := Q(i − 2, T_1) ∪ Q(i − 2, T_2).
    end
    (* Invariant: If T ∈ N, then Q(i − 2, T) is correctly computed.
    Now we use it to compute Q(i, T). *)
    foreach T ∈ N do
        Step 2a: Initialize Q(i, T) := ∅.
        Step 2b: foreach equivalence class S ⊆ Q(i − 2, T) do
            Q(i, T) := Q(i, T) ∪ Growth(S).
        Steps 2c–2d: Compute the equivalence classes of Q(i, T) by finding
            k = 7^3 − 1 nearest neighbors of each i-quad,^a using [9].
        Steps 3–4: Construct faces of the subdivision, by Steps 3–4 in the
            original Build-subdivision, performed on the equivalence classes of Q(i, T).
        if |Q(i, T)| = 1 then delete T from N.
    end
endwhile
```

[a] For each i-quad q, at most $7^3 - 1$ different i-quads $q' \neq q$ can be packed so that $q' \equiv_i q$.

The running time of the L_∞-minimum spanning tree algorithm in [26] is $O(n \log n)$. The k-nearest-neighbors algorithm requires $O(m_i \log m_i + k m_i)$ time to process $m_i = |\mathcal{Q}(i,T)|$ quads in Steps 2c–2d [9]. Since $\sum_i m_i = O(n)$, it takes $O(n \log n)$ total time to perform Steps 2c–2d. We also maintain a disjoint-set data structure to process the $O(n)$ UNION and FIND operations, needed to compute the equivalence classes, efficiently; in any standard

implementation (see, e.g., a survey in [16]) this is not the costliest part of the algorithm. Since the procedure constructs $O(n)$ subfaces, it takes $O(n)$ total time to perform Steps 3–4 for all stages i. Hence, the total running time of the algorithm *Build-subdivision* is $O(n \log n)$. The space requirements of the MST construction in [26] and of the k-nearest-neighbors computation are $O(n)$, as well as the space requirements of the other stages of the algorithm.

We have thus established the following lemma.

Lemma 6.11. *The algorithm* Build-subdivision *can be implemented to run using $O(n \log n)$ standard operations on a real RAM, plus $O(n)$ floor and base-2 logarithm operations.*

Lemmas 6.1, 6.3, 6.4, and 6.11 establish the main theorem of this section.

Theorem 6.12 (Conforming 3D-subdivision Theorem). *Every set of n points in $\mathbb{R}^3$ admits a strongly conforming 3D-subdivision S_{3D} of $O(n)$ size, that also satisfies the minimum vertex clearance property. In addition, each input point is contained in the interior of a distinct whole cube cell. Such a 3D-subdivision can be constructed in $O(n \log n)$ time and linear space.*

7 Extensions and Concluding Remarks

We have presented an optimal-time algorithm for computing an implicit representation of the shortest path map from a fixed source on the surface of a convex polytope in three dimensions. The algorithm takes $O(n \log n)$ preprocessing time and $O(n \log n)$ storage, and answers a shortest path query in $O(\log n)$ time. We have used and adapted the ideas of Hershberger and Suri [22], solving Open Problem 2 of their paper, to construct "on the fly" a dynamic version of the incidence data structure of Mount [32], answering in the affirmative the question that was left open in [32].

As in the planar case (see [22]), our algorithm can also easily be extended to a more general instance of the shortest path problem that involves *multiple sources* on a surface of a convex polytope. Computing shortest paths in the presence of multiple sources is equivalent to computing their (implicit) *geodesic Voronoi diagram*. This is a partition of the polytope surface into regions, so that all points in a region have the same nearest source and the same combinatorial structure (i.e., maximal edge sequence) of the shortest paths to that source. We only compute this diagram implicitly, so that, given a query point $q \in \partial P$, we can identify the nearest source point s to q, and to return the shortest path length (and, possibly, the shortest path itself) from s to q. The algorithm for constructing an implicit geodesic Voronoi diagram is an easy adaptation of the algorithm presented in this paper, with minor (and obvious) modifications. One can show that, for m given sources, the algorithm processes $O(m+n)$ events in total $O((m+n)\log(m+n))$ time, using $O((m+n)\log(m+n))$ storage; afterwards, a nearest-source query can be answered in $O(\log(m+n))$ time.

Finally, we conclude with open problems.

1. Can the space complexity of the algorithm be reduced to linear? Can an efficient

tradeoff between the query time and the space complexity be achieved, using, say, the SPM(s)-representations of Chen and Han [10, 11]?

2. Can an unfolding of a surface cell of S overlap itself?

3. Does the wavefront propagation method extend to the shortest path problem on the surface of a *nonconvex* polyhedral surface? Say, on a polyhedral terrain?

Acknowledgment. We thank Joseph O'Rourke for his thorough review of this paper, as well as for the valuable comments and material on surface unfolding and overlapping, and for remarks on Kapoor's paper. We are also grateful to Haim Kaplan for his help in designing the data structures, and to Joe Mitchell for his comments of Kapoor's paper.

References

[1] P. K. Agarwal, B. Aronov, J. O'Rourke, and C. A. Schevon, Star unfolding of a polytope with applications, *SIAM J. Comput.*, 26:1689–1713, 1997.

[2] P. K. Agarwal, S. Har-Peled, M. Sharir, and K. R. Varadarajan, Approximate shortest paths on a convex polytope in three dimensions, *J. ACM*, 44:567–584, 1997.

[3] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack, An ϵ-approximation algorithm for weighted shortest path queries on polyhedral surfaces, *Abstracts 14th European Workshop Comput. Geom.*, 19–21, 1998.

[4] L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack, An ϵ-approximation algorithm for weighted shortest paths on polyhedral surfaces, *6th Scand. Workshop Algorithm Theory, Lecture Notes Comput. Sci.*, 1432:11–22, Springer-Verlag, 1998.

[5] L. Aleksandrov, A. Maheshwari, and J.-R. Sack, An improved approximation algorithm for computing geometric shortest paths, *14th FCT, Lecture Notes Comput. Sci.*, 2751:246–257, 2003.

[6] G. Aloupis, E. D. Demaine, S. Langerman, P. Morin, J. O'Rourke, I. Streinu, and G. Toussaint, Unfolding polyhedral bands, in *Proc. 16th Canad. Conf. Comput. Geom.*, 60–63, 2004.

[7] B. Aronov and J. O'Rourke, Nonoverlap of the star unfolding, *Discrete Comput. Geom.*, 8:219–250, 1992.

[8] R. Bayer, Symmetric binary B-trees: Data structures and maintenance algorithms, *Acta Informatica*, 1:290–306, 1972.

[9] P. B. Callahan and S. R. Kosaraju, A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields, *J. ACM*, 42(1):67–90, 1995.

[10] J. Chen and Y. Han, Shortest paths on a polyhedron, Part I: Computing shortest paths, *Internat. J. Comput. Geom. Appl.*, 6:127–144, 1996.

[11] J. Chen and Y. Han, Shortest paths on a polyhedron, Part II: Storing shortest paths, Tech. Rept. 161-90, Comput. Sci. Dept., Univ. Kentucky, Lexington, KY, February 1990.

[12] M. de Berg, M. van Kreveld, and J. Snoeyink, Two- and three-dimensional point location in rectangular subdivisions, *J. Algorithms*, 18:256–277, 1995.

[13] E. W. Dijkstra, A note on the problems in connection with graphs, *Numer., Math.*, 1:269–271, 1959.

[14] J. R. Driscoll, D. D. Sleator, and R. E. Tarjan, Fully persistent lists with catenation, *J. ACM*, 41(5):943–949, 1994.

[15] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, Optimal point location in a monotone subdivision, *SIAM J. Comput.*, 15:317–340, 1986.

[16] Z. Galil and G. F. Italiano, Data structures and algorithms for disjoint set union problems, *ACM Computing Surveys*, Vol. 23, Issue 3, 319–344, 1991.

[17] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures — in Pascal and C*, 2nd edition, Addison-Wesley, 1991.

[18] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, Linear time algorithms for visibility and shortest path problems inside simple polygons, *Algorithmica*, 2:209–233, 1987.

[19] L. J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees, in *Proc. 19th IEEE Sympos. Found. Comput. Sci.*, 8–21, 1978.

[20] S. Har-Peled, Approximate shortest paths and geodesic diameters on convex polytopes in three dimensions, *Discrete Comput. Geom.*, 21:216–231, 1999.

[21] S. Har-Peled, Constructing approximate shortest path maps in three dimensions, *SIAM J. Comput.*, 28(4):1182–1197, 1999.

[22] J. Hershberger and S. Suri, An optimal algorithm for Euclidean shortest paths in the plane, *SIAM J. Comput.* 28(6):2215–2256, 1999. Earlier versions: in *Proc. 34th IEEE Sympos. Found. Comput. Sci.*, 508–517, 1993; Manuscript, Washington Univ., St. Louis, 1995.

[23] G. F. Italiano and R. Raman, Topics in Data Structures, in M. J. Atallah, editor, *Handbook on Algorithms and Theory of Computation*, Chapter 5, CRC Press, Boca Raton, 1998.

[24] S. Kapoor, Efficient computation of geodesic shortest paths, in *Proc. 32nd Annu. ACM Sympos. Theory Comput.*, 770–779, 1999.

[25] D. Kirkpatrick, Optimal search in planar subdivisions, *SIAM J. Comput.*, 12:28–35, 1983.

[26] D. Krznaric, C. Levcopoulos, and B. J. Nilsson, Minimum spanning trees in d dimensions, *Nord. J. Comput.*, 6(4):446–461, 1999.

[27] M. Lanthier, A. Maheshwari, and J.-R. Sack, Approximating shortest paths on weighted polyhedral surfaces, *Algorithmica*, 30(4):527–562, 2001.

[28] C. Mata and J. S. B. Mitchell, A new algorithm for computing shortest paths in weighted planar subdivisions, in *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 264–273, 1997.

[29] J. S. B. Mitchell, Shortest paths and networks, in J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry (2nd Edition)*, chapter 27, 607–641, North-Holland, Chapman & Hall/CRC, Boca Raton, FL, 2004.

[30] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, The discrete geodesic problem, *SIAM J. Comput.*, 16:647–668, 1987.

[31] D. M. Mount, On finding shortest paths on convex polyhedra, Tech. Rept., Computer Science Dept., Univ. Maryland, College Park, October 1984.

[32] D. M. Mount, Storing the subdivision of a polyhedral surface, *Discrete Comput. Geom.*, 2:153–174, 1987.

[33] J. O'Rourke, Computational geometry column 35, *Internat. J. Comput. Geom. Appl.*, 9:513–515, 1999; also in *SIGACT News*, 30(2):31–32, (1999) Issue 111.

[34] J. O'Rourke, Folding and unfolding in computational geometry, in *Lecture Notes Comput. Sci.*, Vol. 1763, J. Akiyama, M. Kano, M. Urabe, editors, Springer-Verlag, Berlin, 2000, pp. 258–266.

[35] J. O'Rourke, On the development of the intersection of a plane with a polytope, Tech. Rept. 068, Smith College, June 2000.

[36] J. O'Rourke, S. Suri, and H. Booth, Shortest paths on polyhedral surfaces, Manuscript, The Johns Hopkins Univ., Baltimore, MD, 1984.

[37] R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Massachusetts, 1981.

[38] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[39] M. Sharir, On shortest paths amidst convex polyhedra, *SIAM J. Comput.*, 16:561–572, 1987.

[40] M. Sharir and A. Schorr, On shortest paths in polyhedral spaces, *SIAM J. Comput.*, 15:193–215, 1986.

[41] R. E. Tarjan, *Data Structures and Network Algorithms*, *SIAM CBMS*, 44, 1983.

[42] K. R. Varadarajan and P.K. Agarwal, Approximating shortest paths on a nonconvex polyhedron, in *Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci.*, 182–191, 1997.

[43] E. W. Weisstein, Homotopy, *MathWorld — A Wolfram Web Resource*, `http://mathworld.wolfram.com/Homotopy.html`.

[44] E. W. Weisstein, Riemann Surface, *MathWorld — A Wolfram Web Resource*, `http://mathworld.wolfram.com/RiemannSurface.html`.

[45] E. W. Weisstein, Unfolding, *MathWorld — A Wolfram Web Resource*, `http://mathworld.wolfram.com/Unfolding.html`.

A Artificial waves

In certain cases we can determine that a portion of a one-sided wavefront reaches an edge only after the one-sided wavefront from the other side has swept through the edge. In such cases, we can discard the part that arrives later; we call the resulting wavefronts *approximate one-sided wavefronts*, or simply *approximate wavefronts*. In this sense, an approximate one-sided wavefront is not necessarily a complete representation of all the waves coming from one side of the edge. The approximate wavefront from one side of a transparent edge e is what the true wavefront would be if we were to block off the wavefront from the other side of e by turning e into an *artificial obstacle*. In physical terms, we can imagine replacing the transparent edge e with a high thin wall placed on e, perpendicular to ∂P. The wall absorbs the wavefront from either side of e, although *the wavefront can pass around the endpoints of the wall to reach the other side of* e. Passing around the two endpoints of the wall generates two *artificial waves* that propagate from the two endpoints. The artificial wave is a conceptual device that allows us a limited interaction between the (approximate) wavefronts coming from the two sides of e. See Figure 63(b) for an illustration. This interaction is *too limited to eliminate all the superfluous waves in the pair of the opposite approximate wavefronts at* e, but it suffices to maintain the invariant that *when an approximate wavefront leaves the well-covering region of* e*, it does not contain waves that had already been eliminated from the true wavefront before the time when* e *has been completely covered*. That is, when a wave w is eliminated from the true wavefront before it reaches e, but it does reach e in an approximate wavefront W, w will be eliminated from W by some artificial wave before W leaves $R(e)$. This important invariant is kept due to the well-covering property of e, as described in detail below in this section.

To recap, the approximate wavefront that reaches e from a specific side is the exact wavefront that reaches that side of e in the presence of the obstacle erected at e, and it does not contain waves that were eliminated from the true wavefront before it has entered $R(e)$. We could have calculated the exact wavefront at e explicitly, by merging the two approximate wavefronts from both sides of e, but we do not know how to do it efficiently.

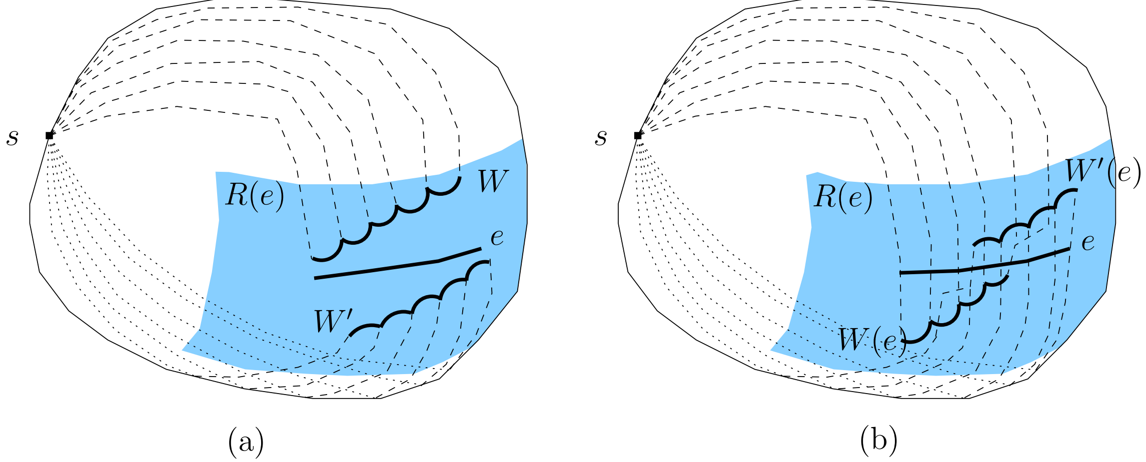

Figure 63: *(a) Two wavefronts W and W', drawn as collections of thick solid arcs, are approaching the transparent edge e from two opposite directions, within the well-covering region $R(e)$ (shaded). The bisectors between the waves are drawn as dashed and dotted lines. (b) Two approximate wavefronts $W(e)$ and $W'(e)$ (collections of thick solid arcs), computed at the simulation time when e is completely covered by W, W', are propagated further within $R(e)$. The algorithm allows for a limited interaction between W and W' in the computation of $W(e)$ and $W'(e)$, which causes some waves that have appeared in W, W' to be removed from $W(e)$ and $W'(e)$, or to be shortened. However, some of the waves that are left in $W(e)$ and $W'(e)$ are obviously absent from the true wavefront, since there is another wave in the opposite approximate wavefront that claims the same points of e.*

The interaction between the opposite one-sided wavefronts is implemented using *artificial waves*. These artificial waves are the only mechanism for pruning portions of the wavefront that arrive second at a transparent edge. We can use the artificial waves to ensure that waves in one approximate wavefront that are dominated by waves from the opposite approximate wavefront are eliminated, within a constant number of cells from where they first become dominated, by one of the artificial waves that they encounter. This can use this property to prove an interesting invariant that is shown in Lemma A.2 below.

The construction is depicted in Figure 64. Consider a transparent edge e, and let a be an endpoint of e (the same is also true for the other endpoint b of e — that is, exactly two artificial waves are constructed at each transparent edge e). Let w_i be the first wave that reaches a. Denote by s_i the generator of w_i, and denote by $\mathcal{E}$ the polytope edge sequence traversed by w_i from s to a. Denote by W' the wavefront that contains w_i ($W' = W(f, e)$ for some edge $f \in \mathit{input}(e)$). Let $W'(e)$ be the approximate wavefront constructed at e to be propagated further through e in the same direction as W', and let $W''(e)$ be the approximate wavefront at e in the opposite direction. We introduce an *artificial wave* with *generator* s_a, which is located at distance $d_S(s, a) = d(s_i, a)$ from a, on the straight line that contains $U_{\mathcal{E}}(e)$ and on the side of $U_{\mathcal{E}}(a)$ that is disjoint from $U_{\mathcal{E}}(e)$ (this part of the line may cross cell boundaries, or boundaries of $R(e)$; there are no visibility constraints for an artificial wave); see Figure 64. To simplify the analysis, this artificial wave is used in the computation of both approximate wavefronts $W'(e)$ and $W''(e)$, even though it is not needed for $W'(e)$; see also a remark below. The triangle inequality implies that $d_S(s, p) \le d_S(s, a) + d_S(a, p)$, for

any point $p \in e$; moreover, since e itself is a shortest path, we have $\left|\overline{U_{\mathcal{E}}(a)U_{\mathcal{E}}(p)}\right| = d_S(a, p)$. Hence, if the artificial wave reaches p before a wavefront W'' from the other side of e reaches p, then p is surely reached first by W', and so there is no need to propagate W'' through p; that is, we construct $W''(e)$ so that it does not contain the portion of W'' that reaches e later than the artificial wave. In essence, an artificial wave is a convenient (and conservative) mechanism for discarding parts of either W' or W'' that can be ascertained to be dominated by the wavefront that reaches e from the other side. The discarded portions are either prefixes or suffixes of the wavefronts. A generator of an artificial wave is not passed on to *output*(e) as part of an approximate wavefront; in other words, an artificial wave at e expands only along e.

Remark: After the elimination of some of the waves approaching e by an artificial wave, an approximate wavefront $W(e)$ may not cover the entire transparent edge e. However, the portions that are not covered by $W(e)$ are necessarily covered by the opposite approximate wavefront.

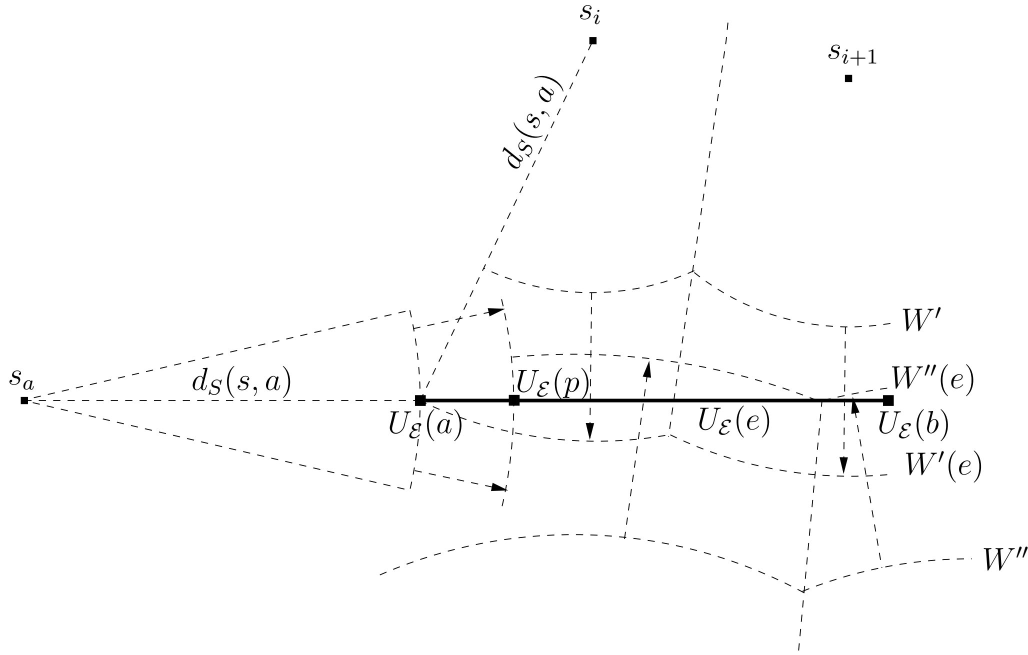

Figure 64: *Two wavefronts, W' and W'', are approaching the transparent edge e from two opposite sides. W' claims the endpoint a before W'', by the wave generated by the source image s_i. The resulting artificial wave is generated by s_a, which is located at distance $d(s_i, a) = d(s, a)$ from a. $W'(e)$ is the approximate wavefront constructed at e to be propagated further through e in the same direction as W', and $W''(e)$ is the approximate wavefront at e in the opposite direction. The point p is the rightmost on e for which the distance $d_S(s, a) + d_S(a, p)$ is less than the time at which the wavefront W'' reaches p, therefore the portion of e between a and p is guaranteed to be reached first by W'; hence the approximate wavefront $W'(e)$ includes the generators of W' that claim this portion of e, while the approximate wavefront $W''(e)$ lacks the corresponding generators of W''.*

Consider the *oriented transparent edge* $\vec{e}$ that coincides with e and is oriented from its endpoint a to its other endpoint b. Consider the computation of the approximate wavefront

at e that will be propagated further (through e) to, say, the right of $\vec{e}$ (that is, reaching $\vec{e}$ from the left; in Figure 64 this is $W'(e)$). The *contributing wavefronts* to this computation are the following:

1. All wavefronts $W(f, e)$, for $f \in input(e)$, that contain at least one wave that reaches $\vec{e}$ from the left (not later than at the time $covertime(e)$).

2. Two artificial waves, one expanding from each endpoint of e.

The contributing wavefronts for the computation of the approximate wavefront reaching $\vec{e}$ from the right are defined symmetrically. Note that a wavefront $W(f, e)$ may contribute to both approximate wavefronts at e (if some wave of $W(f, e)$ reaches e from one side, and another wave of $W(f, e)$ reaches e from the opposite side). The wavefront that crosses no transparent edges on its way from s to e is also contributing for the corresponding one of the approximate wavefronts at e if $s \in R(e)$.

Remark: To simplify the proofs, we say that *both* artificial waves are contributing wavefronts to the approximate wavefront from each side of e. Obviously, it follows from the triangle inequality that an artificial wave with generator s_a cannot actually contribute to the approximate wavefront that claims the endpoint a.

Notice that e is completely covered, including both its endpoints, at the simulation time in which we compute the approximate wavefronts at e. Since, by the invariant we maintain, the exact distance from s to a point on e is the minimum of the distances measured to this point by the two approximate wavefronts, we can compute the exact distances from s to the endpoints of e (see Section 5 for details) and thereby generate exactly the artificial waves that are needed for the computation of the approximate wavefronts at e.

In the following lemma we formalize an important invariant kept by the algorithm, maintained using the artificial waves.

Lemma A.1. *Let s_i be a generator that contributes to an approximate wavefront $W(e)$, but not to the true wavefront at e (because for every point $p \in e$ claimed by s_i, some wave from the other side of e reaches p first). Then the wave of s_i is absent from any approximate wavefront that leaves $R(e)$.*

Proof: Assume the contrary — that is, there is some transparent edge $f \subset \partial R(e)$ so that the approximate wavefront $W(f)$ that has reached f from the side of e includes the wave of s_i. Let q be a point on f claimed by s_i, and p denote the intersection point of $\pi(s_i, q)$ with e. Denote by s_j the true claimer of p (whose wave reaches e from the opposite side). See Figure 65 for an illustration.

Consider first the case in which s is outside $R(e)$; then denote by x the first intersection point of $\pi(s_j, p)$ with $\partial R(e)$. Note that, by definition, x lies on some transparent edge $g \in input(e)$. Then two following cases arise.

First case: If $g = f$, then, since $d_S(q, p) \geq |f|$ (by the well-covering property (W3$_S$)), the endpoints of f are reached by a wave from s_j or from some other generator before the wave from s_j reaches p at time $|\pi(s_j, p)|$. The artificial waves from the endpoints of f will

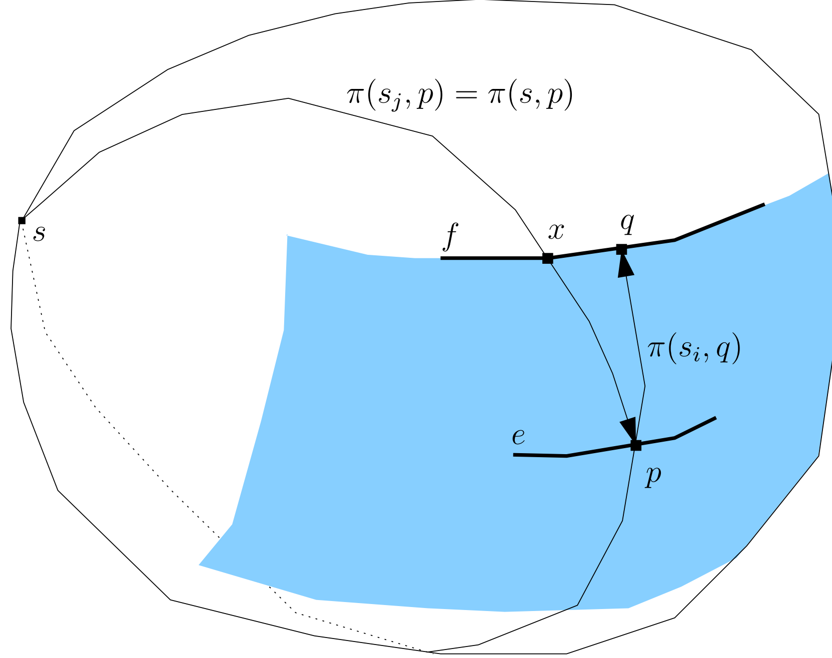

Figure 65: *q lies on f, p lies on e. First case: $\pi(s_j, p)$ intersects f (that is, $g = f$). The well-covering region $R(e)$ is shaded.*

therefore cover f before time $|\pi(s_j, p)| + |f|$. By assumption, we have $|\pi(s_i, p)| > |\pi(s_j, p)|$. The wave from s_i cannot reach e earlier than time $|\pi(s_i, p)| - |e|$. By the well-covering property, $d_S(e, f)$ is at least $|e| + |f|$, and so the wave from s_i reaches f no earlier than $|\pi(s_i, p)| + |f| > |\pi(s_j, p)| + |f|$, at which time f is already covered by the artificial waves. Hence the whole claim of s_i on f is eliminated by the artificial wave, which contradicts the assumption.

Second case: If $g \neq f$, consider the concatenated path $\pi = \pi(s_j, p) \,||\, \pi(p, q)$ that crosses g and reaches f. Since $|\pi(s_j, p)| < |\pi(s_i, p)|$, π reaches q before $\pi(s_i, q)$, from the same side of f. Hence, by Lemma 4.5, there must be a wave w in $W(g, f)$ that is propagated by the algorithm to f (from the same side of f as the wave of s_i) and, during the merging process, does not allow s_i to claim q. Again, this contradicts the assumption.

To complete the proof, we also have to consider the case in which $s \in R(e)$. Then, if $\pi(s_j, p)$ crosses $\partial R(e)$, we can define the intersection point x and the corresponding transparent edge g, and proceed as in the first case above. Otherwise, as in the second case above, the concatenated path $\pi = \pi(s_j, p) \,||\, \pi(p, q)$ reaches q before $\pi(s_i, q)$, from the same side of f. Then again, by Lemma 4.5, this completes the proof. $\square$

Let b be a bisector $b(s_i, s_{i+1})$ of two consecutive source images in some approximate wavefront $W(e)$ of the transparent edge e. We say that in this case b *crosses* e *in* $W(e)$. The following lemma shows that the approximate wavefronts are not too different from the true wavefronts (since, using the artificial waves, each bisector b in the algorithm is not propagated more than $O(1)$ cells away from the vertex of SPM(s) where b ends — see the detailed proof below); this lets us bound the number of the waves that the algorithm propagates.

Lemma A.2. *The number of pairs (e, b) of transparent edges e and bisectors b, such that b crosses e in some approximate wavefront $W(e)$, but b does not cross e in the true wavefront (that is, this crossing does not occur in* SPM(s)*), is $O(n)$.*

Proof: Let X denote the set of these pairs (e, b). Let e be a transparent edge, and let b be a bisector in $W(e)$ so that $(e, b) \in X$. There are only $O(n)$ pairs (e, b) in X in which b is the first or the last (artificial) bisector of $W(e)$, so assume that $b = b(s_i, s_{i+1})$ (so that he waves generated by s_i and s_{i+1} reach e from the same side). Let $p = b \cap e$.

Unless $s \in R(e)$, both s_i, s_{i+1} claim points on $\partial R(e)$ (in *input*(e)) in SPM(s), by Lemma A.1; that is, both paths $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$ intersect $\partial R(e)$. Denote by D_b the region of $R(e)$ between the paths $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$ (if $s \in R(e)$, this area is still well defined, since both paths start from s). Since (e, b) is not an incident pair in SPM(s), there must be at least one bisector event in SPM(s) (that is, a vertex of SPM(s) where b ends) that lies in the interior of D_b. We can charge the early demise of b to any one of these (real) bisector events.

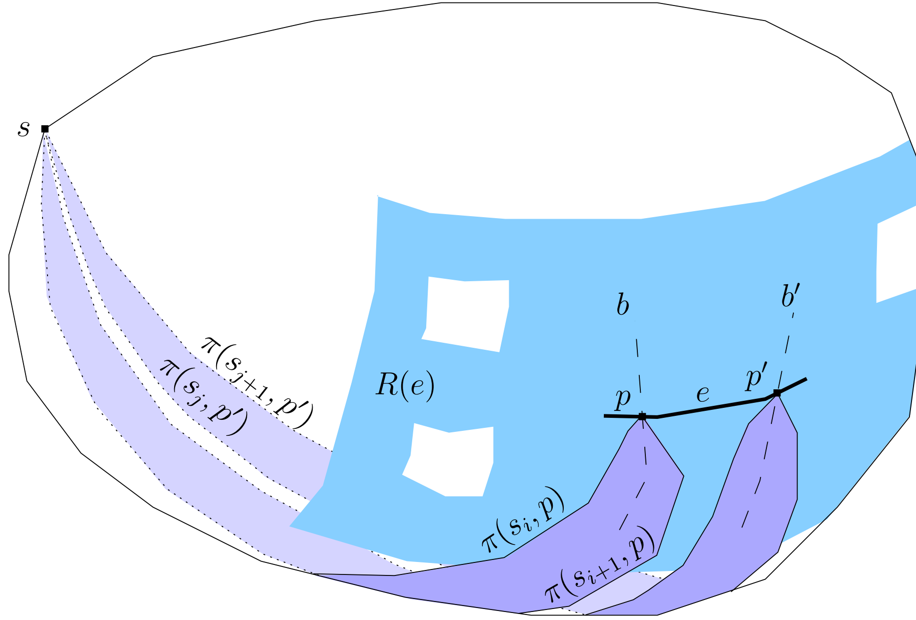

Figure 66: *The bisectors b and b' cross e in the approximate wavefront $W(e)$ (but not in the true wavefront — this fact is not illustrated in this figure). The regions D_b (the sector of $R(e)$ that is bounded by $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$) and $D_{b'}$ (similarly defined) must be disjoint from each other.*

The paths $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$ are disjoint from the corresponding paths defined by any other pair $(e, b') \in X$ — in the modified environment in which e is replaced by a thin high obstacle, the paths $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$ are shortest paths in $\Pi(s, p)$, and hence they are disjoint from any other such paths (see Figure 66 for an illustration). Thus the sector of $R(e)$ that is bounded by $\pi(s_i, p)$ and $\pi(s_{i+1}, p)$ is disjoint from the sector defined by any other pair $(e, b') \in X$, so each vertex v of SPM(s) inside $R(e)$ is charged at most once by all pairs in X that have e as the first element of the pair. Since the surface cell that contains v belongs to only $O(1)$ well-covering regions $R(e)$, v is charged at most $O(1)$ times for all the

relevant pairs in X. Since there are only $O(n)$ vertices in SPM(s) (see Section 2.1.1), this is an upper bound on $|X|$. □

Lemma A.3. *At any simulation time t the total number of waves in all the approximate wavefronts is $O(n)$.*

Proof: Let s_i be a generator in an approximate wavefront at time t, and let e be the last transparent edge that the wave from s_i reaches in the algorithm no later than at t. Consider the interval I claimed by s_i in e. There are $O(n)$ generators that are either the first of the last in their approximate wavefronts, so assume that I is bounded by two bisectors $b(s_{i-1}, s_i)$ and $b(s_i, s_{i+1})$, for two non-artificial generators s_{i-1} and s_{i+1}. Furthermore, we can assume that $b(s_{i-1}, s_i)$ and $b(s_i, s_{i+1})$ both intersect e in SPM(s), recalling that there are only $O(n)$ bisector-edge pairs that appear in some approximate wavefront but not in SPM(s) (by Lemma A.2). Therefore I is bounded by two bisectors of SPM(s), so we can charge any of these bisectors for I. Each bisector of SPM(s) can be charged at most four times, since at most two oppositely directed waves (of distinct approximate wavefronts) can touch each side of the bisector. There are only $O(n)$ bisectors of SPM(s) (see Section 2.1.1), hence the bound follows. □

B Kapoor's algorithm

We briefly describe here the algorithm of Kapoor [24] for computing a shortest path between two points on a general (possibly non-convex) polyhedron P; we also highlight a partial list of the difficulties in the algorithm that remain to be solved in detail to make it possible to validate its correctness and time complexity.

The algorithm follows the continuous Dijkstra paradigm, claiming to compute a shortest path from the source s to a *single target point* t. The algorithm maintains the true wavefront W, propagating the unfolded image of W along the plane ζ that contains some facet f_0 that contains s. Each time that W encounters a facet f of P that was not encountered before, f is unfolded into ζ (each facet is unfolded at most once; some facets may lie in regions where W will be never propagated into, so these facets will never be unfolded — see below). The boundary of the unfolded region consists of disjoined cycles, each cycle encloses a connected region of ∂P whose facets have not been reached by W yet. Only one of these cycles (of edges of ∂P) is maintained by the algorithm — *the cycle B that bounds the region that contains the target point t*. B is subdivided into portions (called *sections* in [24]), each of which is either a single edge b that is *associated* with a sub-wavefront of W that contains the candidate waves to claim points on b, or a sequence $\mathcal{B}$ of edges associated with a single wave $w \in W$ so that w is the best current candidate to claim the whole section $\mathcal{B}$.

The edges of B that are combined into a section $\mathcal{B}$ (and associated with a single wave w) are maintained in a *convex hull tree* structure, so that each node in the tree represents the convex hull of its children. The tree is balanced, so its depth is $O(\log n)$; the structure is claimed to allow to determine the element of $\mathcal{B}$ that is first reached by w in (amortized) $O(\log^2 n)$ time. The waves of W that are associated with a single edge $b \in B$ (that is, the

sub-wavefront $\mathcal{W}$ of claimers of b) are maintained in a similar structure, that is claimed to allow to determine the wave of $\mathcal{W}$ that reaches b first in (amortized) $O(\log^2 n)$ time.

As W is propagated, events are processed; all the events are scheduled in a priority queue. When an event is processed, the data structures must immediately be updated to compute the exact location of the next event. There are several types of possible events:

(i) A wave has been eliminated by its neighbors in W.

(ii) Two non-adjacent waves collide into each other, separating the wavefront into two cycles.

(iii) A wave has reached a facet incident to B that had not been reached before.

(iv) A wave has reached a facet incident to B that had already been reached by another wave.

Events of type (i) are relatively easy to determine (by computing the intersection points of the pair of the bisectors of each wave) and process. However, events of type (ii) are very difficult to detect (using the described data structure), and they are therefore neither detected nor processed by the algorithm. It is claimed in [24] that ignoring these events does not affect the correctness of the algorithm; this claim requires a proof, since a pair of non-adjacent waves that collide into each other might continue advancing "through" each other, possibly affecting the convex hull of the wavefront section that encodes these waves.

Even if we assume that the convex hulls of the wavefront sections are correctly maintained, events of type (iii) are not easy to determine, since the wave w that is closest to B does not have to share points with the convex hull boundary of the wavefront section $\mathcal{W}$ that contains w. The data structure of $\mathcal{W}$ must therefore be efficiently searched to compute the distance from w to B. Although this procedure is sketched in [24], it is not explained how distances along the unfolded surface of P are computed. This is especially problematic when the segment, along which we compute the distance, crosses the boundary of the region unfolded so far onto ζ. This missing detail seems even less trivial if we recall that the unfolded surface of P might overlap itself (see [45]), and the number of faces of ∂P that overlap each other might be large. See Figure 67(a) for an illustration.

The events of type (iv) are even more complicated to process (although, if the wavefront is maintained correctly, they are quite easy to detect). Whenever a wavefront section $\mathcal{W}_1$ (or a wave w_1) reaches a facet that has already been reached by another wavefront section $\mathcal{W}_2$ (or a wave w_2), a *merge* procedure must take place. This procedure has to update three kinds of data structure: (a) it has to merge the data structures that represent the wavefront sections $\mathcal{W}_1, \mathcal{W}_2$; (b) it has to unite the convex hull trees of the sections of B that are associated with the merging waves, and (c) the associations between the edges of B and the merging waves must be updated. While it is sketched in [24] how to perform (c), the description of (a) and (b) does not seem to be complete.

Merging the data structures that represent the wavefront sections (step (a)) does not seem easy, since even two convex hulls C_1, C_2 that comprise k_1 and k_2 arcs, respectively,

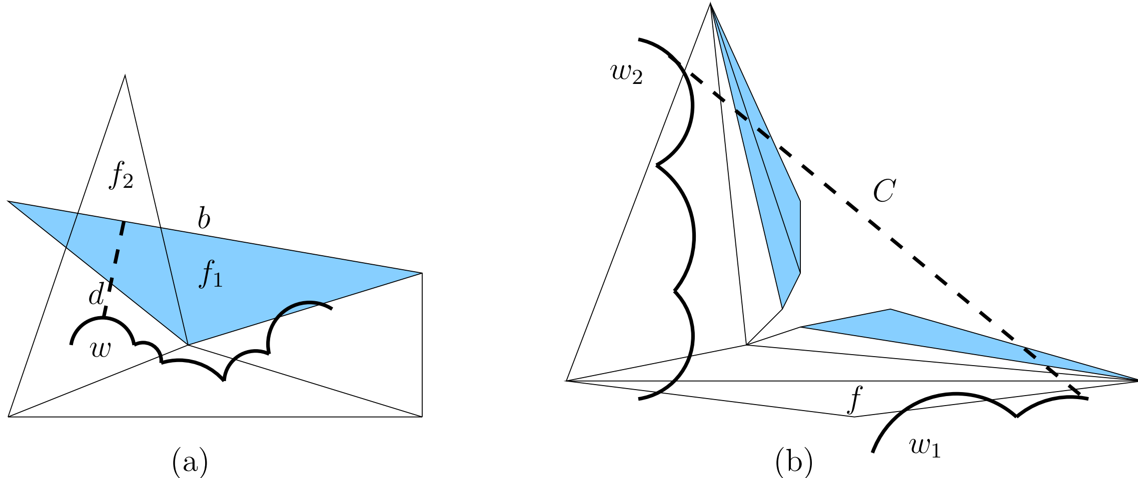

Figure 67: *(a) The unfolded triangle f_1 (shaded), has been currently reached by the wavefront and has been unfolded onto ζ, where its image overlaps another unfolded triangle f_2. The straight line distance d from the wave w to the boundary edge b of f_1 is invalid (the true distance from w to b might be either shorter or longer than d). (b) The wave w_1 reaches the facet f that has already been reached by w_2. The constructed convex hull C intersects the boundary of the unfolded region (facets outside B are shaded).*

might apparently have up to $O(k_1 + k_2)$ intersections. Even if the two convex hulls intersect in only a constant number of points, we recall that these two structures continue to evolve as the corresponding wavefront sections are propagated further along ∂P, and their waves might collide into each other, as in events of type (ii) described above, eventually leading to the intersection of their convex hulls. However, as for the events of type (ii), it is claimed in [24] that ignoring such intersections does not affect the correctness of the algorithm; this probably needs a proof, since it is far from obvious, how can the wavefront data structure, which does not process these events, be queried for shortest distances to B (possibly further complicating the detection of events of type (iii)).

Note also that, as a result of the merging procedure, it is possible for the algorithm to construct a convex hull C of a wavefront section $\mathcal{W}$ so that the region on ζ enclosed in C contains vertices of P that are separated from $\mathcal{W}$ by B — see Figure 67(b) for an illustration. Since the shortest path structure is affected by the way the path "navigates" around the vertices of P, it is unclear in this case how the algorithm determines the right order of the events that occur when $\mathcal{W}$ reaches B, nor how it computes distances within the hull.

Another (probably less significant) missing detail is how the algorithm chooses the cycle that encloses the target point t whenever B is split into more than one cycle (as in step (b) above); recall that the algorithm continues the propagation of W only towards the cycle that contains the target t, neglecting the portions of W that are not associated with that cycle. It is easy to construct an example where B is split $\Theta(n)$ times through the algorithm; moreover, even during a single merge the boundary might be split into many cycles. Therefore, the correct determination of the cycle that encloses t (which does not

seem to be simple, especially in the case of a nonconvex polytope) is important to estimate the algorithm running time. (It is conceivable that some lock-step searching mechanism could handle this problem, but the details are not obvious to us.)

To summarize, as it is presented, we feel that the algorithm of Kapoor [24] has many issues to address and to fill in before it can be judged at all.