Discrete Comput Geom (2008) 39: 500–579
DOI 10.1007/s00454-007-9031-0

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

Yevgeny Schreiber · Micha Sharir

Received: 19 January 2007 / Revised: 4 August 2007 /
Published online: 19 September 2007
© Springer Science+Business Media, LLC 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 (Hershberger, J., Suri, S. in SIAM J. Comput. 28(6):2215–2256, 1999), 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 introduced by Hershberger and Suri 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 (Mount, D.M. in Discrete Comput. Geom. 2:153–174, 1987) 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.

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.

Y. Schreiber (✉) · M. Sharir
School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
e-mail: syevgeny@tau.ac.il

M. Sharir
Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA
e-mail: michas@tau.ac.il

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 $O(\log(n+m))$.

Keywords Continuous Dijkstra · Geodesics · Polytope surface · Shortest path · Shortest path map · Unfolding · Wavefront

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, or, more generally, computing a compact representation of all such paths that emanate from a fixed source point s, is a classical problem in geometric optimization, first studied by Sharir and Schorr [36]. Their algorithm, whose running time is $O(n^3 \log n)$, constructs a planar layout of the *shortest path map*, and then the length and combinatorial type of the shortest path from s to any given query point q can be found in $O(\log n)$ time; the path itself can be reported in $O(k)$ additional time, where k is the number of edges of P that are traversed by the shortest path from s to q. Soon afterwards, Mount [27] gave an improved algorithm for convex polytopes with running time $O(n^2 \log n)$. Moreover, in [28], 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 polyhedron P, O'Rourke et al. [31] gave an $O(n^5)$-time algorithm for the single source shortest path problem. Subsequently, Mitchell et al. [26] presented an $O(n^2 \log n)$ algorithm, extending the technique of [27]. All algorithms in [26, 27, 36] use the same general approach, called "continuous Dijkstra", first formalized in [26]. 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. The approach treats certain elements of ∂P (vertices, edges, or other elements) as nodes in a graph, and follows Dijkstra's algorithm 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 [8] 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 [8] also constructs a planar layout of the shortest path map (which is "dual" to the layout of [36]), 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 [6], that this layout of P does not overlap itself.) In [9], Chen

and Han follow the general idea of Mount [28] to solve the problem of storing shortest path information separately, for a general, possibly nonconvex polyhedral surface. They obtain a tradeoff between query time $O(d \log n / \log d)$ 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 [2–4, 16, 17, 19, 24, 25, 38] presented approximate algorithms for the 3-dimensional shortest path problem. Nevertheless, the major problem of obtaining a subquadratic, or even near-linear, exact algorithm remained open. In 1999, Kapoor [21] 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 [29]). The algorithm follows the continuous Dijkstra paradigm, and claims to be able to compute a shortest path *between two given points* in $O(n \log^2 n)$ time (so it does not preprocess the surface for answering shortest path queries). However, as far as we know, the details of Kapoor's algorithm have not yet been published.

The Algorithm of Hershberger and Suri for Polygonal Domains A dramatic breakthrough on a loosely related problem took place in 1995,[1] when Hershberger and Suri [18] obtained an $O(n \log n)$-time algorithm for computing shortest paths *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.

Our algorithm uses (adapted variants of) many of the ingredients of [18], including the continuous Dijkstra method—in [18], the wavefront is propagated amid the obstacles, where each wave emanates from some obstacle vertex already covered by the wavefront; see Fig. 1(a).

The key new ingredient in [18] is a quad-tree-style subdivision of the plane, of size $O(n)$, on the vertices of the obstacles (temporarily ignoring the obstacle edges).

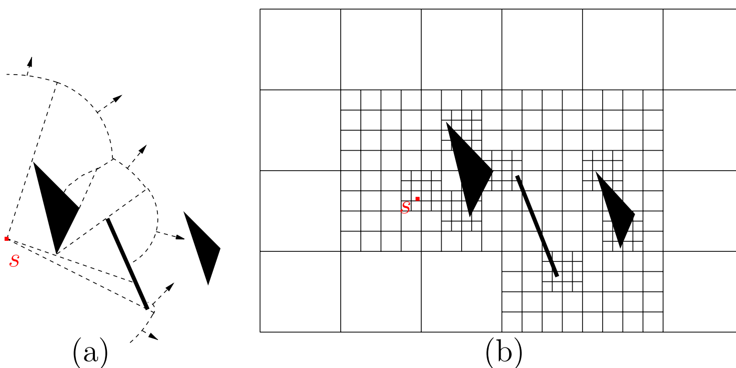

Fig. 1 The planar case: (**a**) The wavefront propagated from s, at some fixed time t. (**b**) The conforming subdivision of the free space

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

See Fig. 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 is "simple," 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.

1.2 An Overview of Our Algorithm

As in [18], 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 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. 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 process any such unfolding query in $O(\log n)$ time. This is a nontrivial addition to the machinery of [18] (where the transparent edges are simply straight segments, which are trivial to represent and to manipulate).

However, in order to propagate the wavefront along the surface of P, we have to overcome another difficulty. On top of the main problem that a surface cell may intersect many (up to $\Theta(n)$) facets of P, it can in general be unfolded in more than one way, and such an unfolding may *overlap* itself (see [11]). To overcome this, 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.

We maintain two *one-sided wavefronts* instead of one exact wavefront at each transparent edge e, so 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. At each step of the wavefront propagation phase, the algorithm 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 [18]. Each propagation that reaches f from e proceeds along a fixed sequence of building blocks that connect e to f. 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.

A key difference from [18] 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 by itself 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.

During each propagation, we keep track of combinatorial changes that occur *within* the wavefront: 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 [18], we show that the algorithm encounters a total of only $O(n)$ "events," and processes each event in $O(\log n)$ 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 [18], 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. The query "identifies" the path combinatorially. It can immediately produce 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 to compute, where k is the number of polytope edges crossed by the path.

To aid readers familiar with [18], the structure of our paper closely follows that of [18], although each part that corresponds to a part of [18] is quite different in technical details. 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 Sect. 6 (it is nevertheless very similar to its counterpart in [18], and we only describe the differences between the two procedures). The construction in Sect. 2 is new and involves many ingredients that cater to the spatial structure of convex polytopes. Section 3 also has no parallel in [18]—it presents the Riemann structure, which represents the unfolding of the polytope surface, as needed for the

[2] An explicit representation is tricky in any case, because the map, in its folded form, has quadratic complexity in the worst case.

implementation of the wavefront propagation phase. Section 4 describes the wavefront propagation phase 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 Sect. 5. We close in Sect. 7 with a discussion, which includes the extension to the construction of *geodesic Voronoi diagrams* on ∂P, and with several open problems.

The full version of the paper [34] is even longer than this journal version—it builds upon the already long paper [18], and adds many new technical steps in full detail. This shorter journal version contains most of its ingredients, but omits certain steps, such as those sufficiently similar to their counterparts in [18].

2 A Conforming Surface Subdivision

A key ingredient of the algorithm is a special subdivision S of ∂P, which we construct in two steps. The first step, sketched in Sect. 6, builds a rectilinear oct-tree-like subdivision S_{3D} of $\mathbb{R}^3$ by taking into account only the vertices of P (see [34, Sect. 6] for details). In the present section, we only state the properties that S_{3D} should satisfy, assume that it is already available, and describe the second step, which constructs S from S_{3D}. We start with some preliminary definitions.

2.1 Preliminaries

Without loss of generality, we assume that s is a vertex of P, that all facets of P are triangles, and that no edge of P is axis-parallel. Our model of computation is the real RAM.

We borrow some definitions from [26, 35, 36]. A *geodesic path* π is a simple path along ∂P so 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 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 [26]. We use the notation $\Pi(a, b)$ to denote the set of all 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 $d_{3D}(x, y)$ (resp., $d_\infty(x, y)$) to denote the Euclidean (resp., the L_∞) distance in $\mathbb{R}^3$ between x, y; when considering points x, y on a plane, we sometimes denote $d_{3D}(x, y)$ by $d(x, y)$.

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 $\Phi_{\mathcal{E}} = \varphi_k \circ \varphi_{k-1} \circ \cdots \circ \varphi_1$. (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 $U_{\mathcal{E}}$. Since each rigid transformation in $\mathbb{R}^3$ can be represented as a 4×4 matrix [32] (see [34] for details), 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.

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).[3] 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 $U_{\mathcal{E}}$.

The following properties of shortest paths are proved in [8, 26, 35, 36]: (i) The intersection of a shortest path 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) Two shortest 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.

The Elements of the Shortest Path Map 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 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 peel Φ, there is only one shortest path $\pi(s, p) \in \Pi(s, p)$ to any $p \in \Phi$, which also satisfies $\pi(s, p) \subset \Phi$.

There are two types of intrinsic vertices of SPM(s) (excluding intersections of peel boundaries with edges of P): ridge points that are incident to three or more peels, and vertices of P (including s). The boundaries of the peels form the *edges* of SPM(s). There are two types of edges (see Fig. 2): (i) shortest paths from s to a vertex of P, and (ii) *bisectors*, each being a maximal connected polygonal path of ridge points between two vertices of SPM(s) that does not contain any vertex of SPM(s).

It is proved in [36] that: (1) A shortest path from s to any point in ∂P cannot cross a bisector. (2) SPM(s) has only $O(n)$ vertices and (folded) edges, each of which is a union of $O(n)$ straight segments.

[3] 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.

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 ($\mathcal{E}_i$ is unique, since Φ_i 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 also denote by s_i the unfolded source image $U_{\mathcal{E}'_i}(s)$, where $\mathcal{E}'_i$ is some prefix 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 respective corresponding facet sequences of $\mathcal{E}_i, \mathcal{E}_j$.

2.2 The 3-Dimensional Subdivision and Its Properties

We begin by introducing the subdivision S_{3D} of $\mathbb{R}^3$, whose construction is sketched in Sect. 6. The subdivision is composed of 3D-cells, each of which is an axis-parallel cube, either whole, or *perforated* by a single axis-parallel cube-shaped hole;[4] see Fig. 3. The boundary face of each 3D-cell is divided into either 16×16 or 64×64 square subfaces with axis-parallel sides.

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:

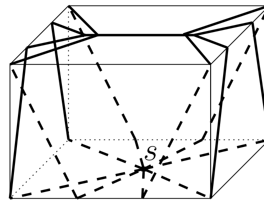

Fig. 2 Peels are bounded by *thick lines* (*dashed and solid*). The bisectors (the set of all the ridge points) are the *thick solid lines*, while the *dashed solid lines* are the shortest paths from s to the vertices of P

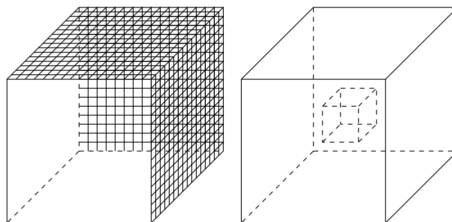

Fig. 3 Two types of a 3D-cell: a whole cube (where the subdivision of three of its faces is shown), and a perforated cube (it is not shown here that each of its inner and outer faces is subdivided into subfaces)

[4]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.

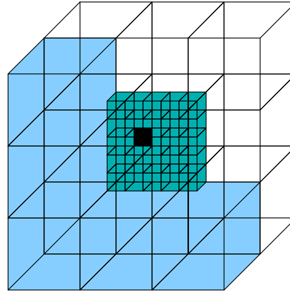

Fig. 4 The well-covering region of the *darkly shaded* face h contains, in this example, a total of 39 3D-cells (nine transparent large cells on the back, five *lightly shaded* large cells on the front, and 25 small cells, also on the front). Each face of the boundary of each 3D-cell in this figure is further subdivided into subfaces (not shown). The well-covering region of each of the subfaces of h coincides with $R(h)$

(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 Fig. 4).

(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:[5]

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

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.

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 sketched in Sect. 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*

[5]The wavefront propagation algorithm described in Sects. 4 and 5 requires the subfaces of S_{3D} only to be well-covered, but not necessarily strongly well-covered. The stronger condition (W3′) of subfaces of S_{3D} is needed only in the construction of the surface subdivision S.

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

minimum vertex clearance property. In addition, each input point is contained in the interior of a distinct whole cube cell. Such a 3*D-subdivision can be constructed in* $O(n \log n)$ *time.*

2.3 Computing the Surface Subdivision

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 (transparent) edge* e of S with the same pair of endpoints. (We textitasize here that $e \neq \xi$. 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 Fig. 5). 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*.

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, $U_{\mathcal{E}}(e_{a,b})$ is not necessarily a straight segment, but we will shortly show that it is); see Fig. 6. We say that $e_{a,b}$ *originates from the cut* $\xi(a,b)$. Obviously, its length $|e_{a,b}|$ is equal to $|U_{\mathcal{E}}(e_{a,b})| \leq |\xi(a,b)|$. (This initial collection of transparent edges may contain crossing pairs, and

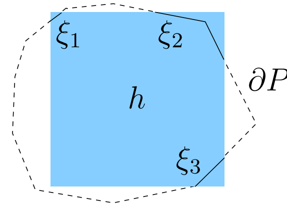

Fig. 5 A subface h and three maximal connected portions ξ_1, ξ_2, ξ_3 that constitute the intersection $h \cap \partial P$

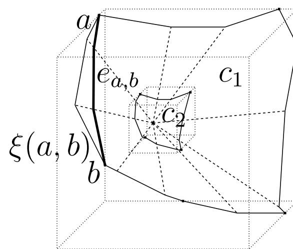

Fig. 6 The cuts of the boundaries of the 3D-cells c_1, c_2 with ∂P are denoted by *thin solid lines*, and the *dashed lines* denote polytope edges. The transparent edge $e_{a,b}$ that originates from the cut $\xi(a,b)$ is *bold*. (To simplify the illustration, this figure ignores the fact that the faces of S_{3D} are actually subdivided into smaller subfaces)

each initial transparent edge will be split into sub-edges at the points where other edges cross it—see below.)

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 (MVC), for any subface h of $S_{3\mathrm{D}}$ and for any $v \in V$, we have $d_{3\mathrm{D}}(h, v) \geq 4l(h)$. 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 $d_{3\mathrm{D}}(a, v) \geq |e_{a,b}|$, which shows that $e_{a,b}$ cannot reach any vertex v of P. □

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

Proof Easy, and omitted; it follows from the structure of $S_{3\mathrm{D}}$. □

Lemma 2.4 *Each transparent edge that originates from some face ϕ of $S_{3\mathrm{D}}$, meets at most $O(1)$ other transparent edges that originate from faces of $S_{3\mathrm{D}}$ adjacent to ϕ (or from ϕ itself), and does not cross any other transparent edges (which originate from faces of $S_{3\mathrm{D}}$ 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 $S_{3\mathrm{D}}$ that contain $\xi(a, b)$ and $\xi(c, d)$, respectively. Since $a, b \in h$, we have $d_{3\mathrm{D}}(e_{a,b}, h) < \frac{1}{2}|e_{a,b}| \leq \frac{1}{2}|\xi(a, b)| \leq 2l(h)$. Similarly, $d_{3\mathrm{D}}(e_{c,d}, g) \leq 2l(g)$. Recall that $S_{3\mathrm{D}}$ is a strongly conforming 3D-subdivision. Therefore, if h, g are incident to non-adjacent faces of $S_{3\mathrm{D}}$, then, by (W3′), $d_{3\mathrm{D}}(h, g) \geq 16 \max\{l(h), l(g)\}$, hence $e_{a,b}$ does not intersect $e_{c,d}$. Since are only $O(1)$ faces of $S_{3\mathrm{D}}$ that are adjacent to the face of h, and each of them contains $O(1)$ subfaces g, there are at most $O(1)$ possible choices of g for each h. □

Splitting Intersecting Transparent Edges Crossing transparent edges are illustrated in Fig. 7. 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.

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 Fig. 8).*

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 single triangle, so $k \geq 1$. Since e and e' are shortest paths

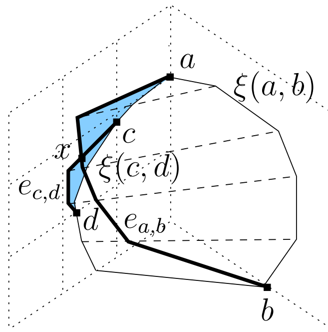

Fig. 7 Subfaces are bounded by *dotted lines*, polytope edges are *dashed*, the cuts of $\partial P \cap S_{3D}$ are *thin solid lines*, and the two transparent edges $e_{a,b}, e_{c,d}$ are drawn as *thick solid lines*. The edges $e_{a,b}, e_{c,d}$ intersect each other at the point $x \in \partial P$; the *shaded region* of ∂P (including the point x on its boundary) lies in this illustration beyond the plane that contains the cut $\xi(c, d)$

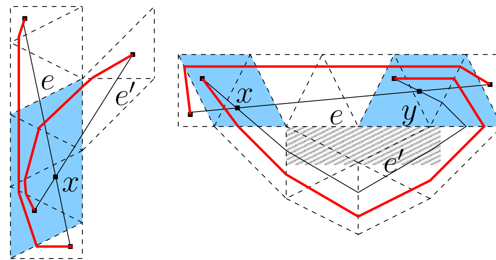

Fig. 8 Two examples of intersecting transparent edges e, e' (*thin solid lines*); the corresponding original cuts (*thick solid lines*) never intersect each other. The maximal contiguous facet subsequences that are traversed by both e, e' and contain an intersection point of $e \cap e'$ are *shaded*. In the second example, the "hole" of ∂P between the facet sequence traversed by e and the facet sequence traversed by e' is hatched

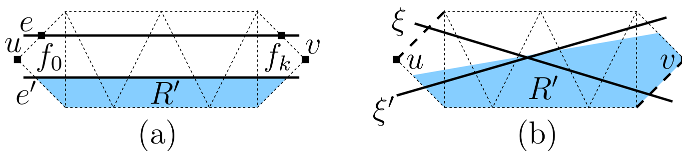

Fig. 9 (**a**) e' divides R into two regions, one of which, R' (*shaded*), contains neither u nor v. (**b**) If R' contains v but not u, ξ' (crossing the same edge sequence as e') intersects ξ (which must cross the bold dashed edges, since R is maximal)

within R, they cannot cross each other (within R) more than once, which proves the first part of the lemma.

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 Fig. 9. 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. 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* 3*D-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* c, *all transparent edges on* ∂c *are induced by* $O(1)$ 3*D-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. □

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 called the *well-covering region* of e.

(W2$_S$) The total number of transparent edges in 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.

(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 admits a conforming surface subdivision S into $O(n)$ transparent edges and surface cells, constructed as described above.*

Proof The properties (C1$_S$), (C3$_S$) follow from the properties (C1), (C3) of $S_{3\mathrm{D}}$, 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 $S_{3\mathrm{D}}$ 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 that is well-covered in $S_{3\mathrm{D}}$, 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 $S_{3\mathrm{D}}$ 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) $R(h)$, $d_S(h, g) \geq d_{3\mathrm{D}}(h, g) \geq 16\max\{l(h), l(g)\}$. 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 $d_{3\mathrm{D}}(p, h) \leq 2l(h)$, and for each $q \in e'$ we have $d_{3\mathrm{D}}(q, g) \leq 2l(g)$. Hence, for each $p \in e, q \in e'$, we have $d_S(p, q) \geq d_{3\mathrm{D}}(p, q) \geq (16 - 4)\max\{l(h), l(g)\}$, and therefore $d_S(e, e') \geq 2\max\{|e|, |e'|\}$. □

We next simplify S by deleting (all the transparent edges of) 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 Fig. 10. (This optimization clearly does not violate any of the properties of S proved above.) After the optimization, each hole of a surface cell of S must contain a vertex.

The following lemma sharpens a simple property of S that is used later in Sect. 3.

Lemma 2.10 *A transparent edge e intersects any polytope edge in at most one point.*

Proof A polytope edge χ can intersect e at most once, since e is a shortest path (within the union of a facet sequence); since we assume that no edge of P is axis-parallel, $e \cap \chi$ cannot be a nontrivial segment. □

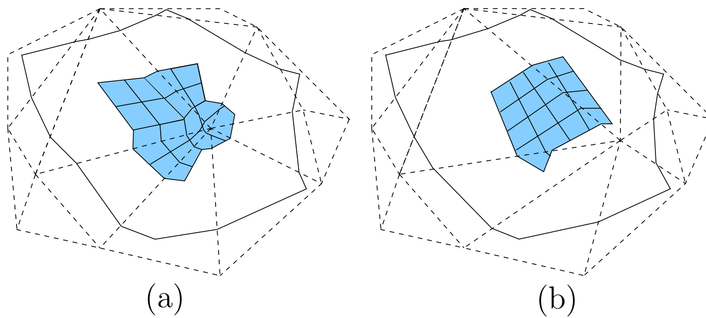

Fig. 10 Simplifying the subdivision (dashed edges denote polytope edges, and solid edges denote transparent edges). (**a**) None of the cells is discarded, since, although the *shaded cells* are completely contained inside a single hole of another cell, one of them contains a vertex of P. (**b**) All the *shaded cells* are discarded, and become part of the containing cell

2.4 The Surface Unfolding Data Structure

In this subsection we present the *surface unfolding data structure*, which we define and use to efficiently construct the surface subdivision. This data structure is also used in Sect. 3 to construct more complex data structures for wavefront propagation and in Sect. 5 by the wavefront propagation algorithm.

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 (when viewed from above) 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 [20], 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 can use T_z to perform the following type of query: Given a horizontal subface $h = [a, b] \times [c, d] \times \{z_1\}$ of S_{3D}, compute efficiently the convex polygon $P \cap h$, and represent its boundary in compact form (without computing $P \cap h$ explicitly). We access the value $T_z(z_1)$ of T_z at $z = z_1$ (which represents $P(z_1)$), and compute the intersection points of each of the four 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.

We can also use T_z for another (simpler) type of query: Given a facet f of ∂P and some $z = z_1$, locate the endpoints of $f \cap P(z_1)$ (which must be stored at two consecutive leaves in the cyclic order of leaves of the corresponding version of T_z), or report that $f \cap P(z_1) = \emptyset$. 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 $P(z_1)$ 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 $P(z_1)$, in $O(\log n)$ time, by a binary search in the sequence of their slopes. To make binary search possible in $O(\log n)$ time (as well as to enable a somewhat more involved search over T_z that we use in the proof of Lemma 3.12), we store at each node of T_z a pair of pointers to the rightmost and leftmost leaves of its subtree. These extra pointers can be easily maintained during the insertions to and deletions from T_z; it is also easy to see that updating these pointers is coherent with the path-copying method.

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, 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 affect the asymptotic time complexity; maintaining the unfolding information while rebalancing the tree can be performed in a manner similar to that used in another related data structure, described in Sect. 5.1, with full, and fairly routine, details given in [34].) 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.

Let $\mathcal{F} = (f_0, f_1, \ldots, f_k)$ denote the corresponding facet sequence of the sequence of 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

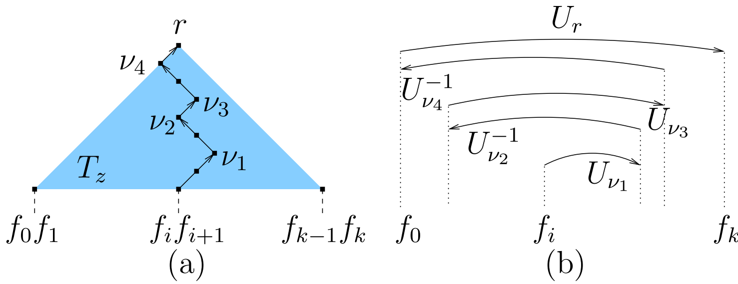

Fig. 11 Constructing U_i by traversing the path from the polytope edge succeeding the facet f_i to the root r of T_z. (**a**) The nodes ν_1, ν_3 are the left turns, and the nodes ν_2, ν_4 are the right turns in this example. (**b**) Composing the corresponding transformations stored at $\nu_1, \dots, \nu_4$ and at r

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 Fig. 11 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.

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.

Hence we can compute, in $O(\log n)$ time, the image of any 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. 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* 3*D-subdivision* $S_{3\mathrm{D}}$, *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, for each subface h of $S_{3\mathrm{D}}$, 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 Sect. 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 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 $S_{3\mathrm{D}}$ that induce e, e'). Denote by $\mathcal{F} = (f_0, \dots, f_k)$ (resp., $\mathcal{F}' = (f'_0, \dots, f'_{k'})$) 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. □

3 Surface Unfoldings and Shortest Paths

In this section we show how to unfold the surface cells of S and how to represent these unfoldings for the wavefront propagation algorithm (described in Sects. 4 and 5) as *Riemann structures*. Informally, this representation consists of unfolded "flaps," which we call *building blocks*, all lying in a common plane of unfolding. We glue them together locally without overlapping, but they may globally have some overlaps, which however are ignored, since we consider the corresponding flaps to lie at different "layers" of the unfolding.

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 Fig. 12(a). We define $G(e, e')$ to be the collection of all *maximal* connecting common subsequences of $\mathcal{E}$ and $\mathcal{E}'$.

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 Fig. 12(b).

In either case, the elements of such a collection $G(x, y)$ do not share any polytope edge. We say that a subsequence in $G(x, y)$ *connects* x and y.

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 $\mathcal{E}_{x,y} = (e_0, e_1, \ldots, e_k) \in G(x, y)$ 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. Define the *shortened facet sequence* of $\mathcal{E}_{x,y}$ to be

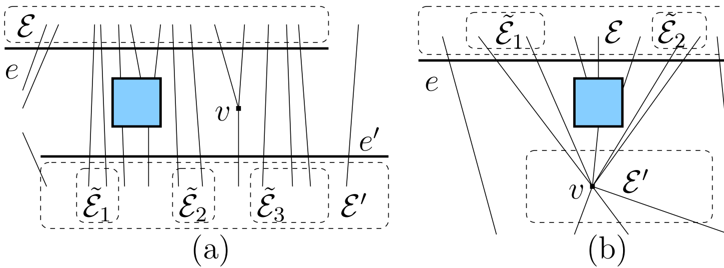

Fig. 12 Maximal connecting common subsequences of polytope edges (drawn as thin solid lines) in (**a**) $G(e, e')$, and (**b**) $G(e, v)$. The transparent edges are drawn thick, and the interiors of the transparent boundary edge cycles that separate $\tilde{\mathcal{E}}_1$ and $\tilde{\mathcal{E}}_2$ are *shaded*

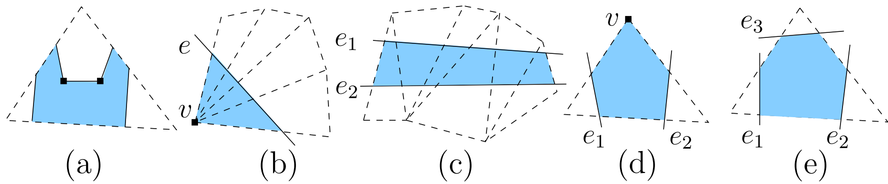

Fig. 13 Building blocks (*shaded*): (**a**), (**b**), (**c**) of types I, II and III, respectively, and (**d**), (**e**) of type IV

$\mathcal{F} \setminus \{f_0, f_k\}$ (so that the extreme edges e_0, e_k of $\mathcal{E}_{x,y}$ are on the boundary of its union), and note that the shortened sequence can be empty (when $k = 1$). We define the following four types of *building blocks* of c.

Type I: Let f be a facet of ∂P. 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 Fig. 13(a) for an illustration.

Type II: Let v be the unique vertex in $V(c)$ (assuming it exists), e a transparent edge in ∂c, and $\mathcal{E}_{e,v} \in G(e, v)$ a maximal subsequence connecting 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; see Fig. 13(b).

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; see Fig. 13(c).

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 Fig. 13(d), (e).

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

$\mathcal{E}_1 = (\chi_2, \ldots, \chi_{k-1})$, $\mathcal{E}_2 = (\chi_{k-1}, \ldots, \chi_2)$. Note that neither $\mathcal{E}_1$ nor $\mathcal{E}_2$ is cyclic, and that the unfolded images $U_{\mathcal{E}_1}(B)$, $U_{\mathcal{E}_2}(B)$ are congruent.

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 $U_{\mathcal{E}}(x) = U_{\mathcal{E}}(y)$. The following lemma states an important property of building blocks (which easily follows from their definition).

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 Easy, and omitted. □

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 (see Fig. 13(d)), or (b) B is a convex hexagon, whose boundary alternates between portions of the edges of f and portions of transparent edges (see Fig. 13(e)). In the latter case, B contains no vertices of P (i.e., of f).*

Proof Easy, and omitted. □

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 construction. If B is of type IV, then by Lemma 3.2, $U_{\mathcal{E}}(B) = B$ is a convex pentagon or hexagon. If B is of type III, then $U_{\mathcal{E}}(B)$ is a convex quadrilateral, by construction. □

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 x can be incident to at most one building block of c of type I (or to at most two such blocks, if our general position assumption is not strong enough—in that case x may be incident to an edge, but not to a vertex, of P).

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 vertex of P. Since there are at most $O(1)$ transparent edge cycles in ∂c 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

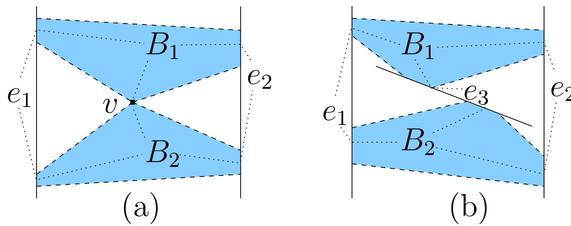

Fig. 14 The triple, of (**a**) two transparent edges and a vertex of P, or (**b**) three transparent edges, contributes to two building blocks B_1, B_2. The corresponding graphs $K_{3,2}$ are illustrated by dotted lines. If the triple contributed to three building blocks, we would have obtained an impossible plane drawing of $K_{3,3}$

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.

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 Fig. 14). 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 Fig. 14), 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. □

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

Proof 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; the proof of this claim is easy, and omitted. □

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 Fig. 13 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, III, or IV, the claim is immediate. □

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 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: (1) 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 Fig. 15(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). (2) 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 Fig. 15(b). Then (χ_1, χ_2) connects

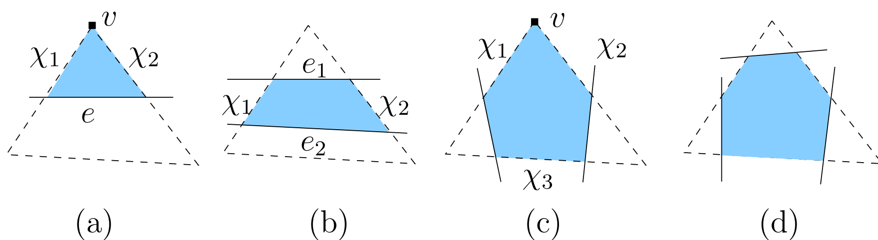

Fig. 15 If Q (shaded) does not contain a transparent endpoint, it must be either a portion of a building block of (**a**) type II or (**b**) type III, or (**c**), (**d**) a building block of type IV

e_1 and e_2, hence p is in a building block of type III (again, f lies in the shortened facet sequence). (3) 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 Fig. 15(c). Then p lies in a building block of type IV. (4) Q is a hexagon bounded by all three edges of f and by three transparent edges. See Fig. 15(d). Again, by definition, p lies in a building block of type IV. This (and the disjointness of building blocks established in Lemma 3.6) completes the proof of the lemma. □

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

Lemma 3.10 *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 compute the boundary of each such block by a straightforward iterative process that starts at a transparent endpoint a lying in some facet f of P, and traces the block boundary from a along an alternating sequence of transparent edges and edges of f (with the possible exception of traversing, once, two consecutive edges of f through a common vertex), until we get back to a.

Since, by Corollary 3.4, there are no holes inside building blocks, after each boundary tracing step 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 unfolding $O(1)$ transparent edges in $O(\log n)$ time, using the surface unfolding data structure. (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.) □

Lemma 3.11 *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 3.10. Denote by $\mathcal{H}$ the set of facets in $\mathcal{F}_v$ that contain building blocks of c of type I *that are incident to v*. Denote by $\mathcal{Y}$ the set of maximal contiguous subsequences that constitute $\mathcal{F}_v \setminus \mathcal{H}$. To compute $\mathcal{Y}$, we locate each facet of $\mathcal{H}$ 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 $\mathcal{Y}$ iteratively. Each step picks a nonempty sequence $\mathcal{F} \in \mathcal{Y}$ 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 $\mathcal{Y}$. Since there are no cyclic transparent edges, by construction, it easily follows that $\mathcal{H} \cap \mathcal{F}_v \neq \emptyset$, and therefore $\mathcal{F}$ is not cyclic. Denote the facets of $\mathcal{F}$ by $f_1, \ldots, f_k$, with $k \geq 1$. Denote by $(\chi_1, \ldots, \chi_{k-1})$ the corresponding

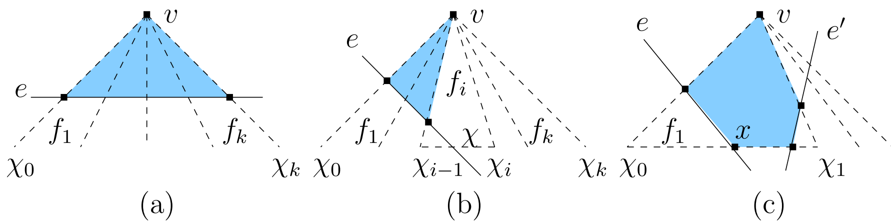

Fig. 16 Extracting from $\mathcal{F}$ building blocks (drawn *shaded*) of type II (cases (**a**), (**b**)) or IV (case (**c**))

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). Note that $\mathcal{F}$ cannot be interrupted by a hole in c, since the endpoints of the transparent edges of such a hole lie in blocks of type I, which belong to $\mathcal{H}$.

In the former case (see Fig. 16(a)), mark the region of ∂P between e, χ_0, and χ_k as a building block of type II, delete $\mathcal{F}$ from $\mathcal{Y}$, and terminate this iteration of the loop. In the latter case, there are two possible cases. If $i > 1$ (see Fig. 16(b)), mark the region 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 $\mathcal{H}$. See Fig. 16(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 $\mathcal{Y}$, 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 3.12 *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 3.10 and 3.11. Denote the set of all these building blocks by $\mathcal{H}$. (Note that $\mathcal{H}$ 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 $\mathcal{H}$. 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 $\mathcal{H}$ 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 $\mathcal{H}$. Each contact interval in L bounds two building blocks of c, one of which is in $\mathcal{H}$ (it is either of type I or contains a vertex of P in its closure), and the other is not in $\mathcal{H}$ 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 $\mathcal{H}$ consists of several connected components. Since there are no blocks of $\mathcal{H}$ 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 $\mathit{limits}(I)$ the pair of transparent edges that delimit it.

Denote by $\mathcal{Y}$ 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 $\mathcal{Y}$ appears in a unique cycle. Since there are only $O(1)$ building blocks of c, we can compute the sequences of $\mathcal{Y}$ in constant time. Let $Y = (I_1, I_2, \dots, I_k)$ be a cyclic sequence in $\mathcal{Y}$ (with $I_{zk+l} = I_l$, for any $l = 1, \dots, k$ and any $z \in \mathbb{Z}$). Then, for every pair of consecutive intervals $I_j, I_{j+1} \in Y$, $\mathit{limits}(I_j) \cap \mathit{limits}(I_{j+1})$ 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 $\mathcal{Y}$ contains two or more contact intervals. As argued above, the portion of ∂P bounded by these contact intervals and by their connecting transparent edges is a portion of c which consists of only building blocks of types III and IV. In particular, it does not contain in its interior any vertex of P, nor any transparent edge.

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

If Y contains exactly two contact intervals, they must bound a single building block of type III, which we can easily compute, and then discard Y. Otherwise, let I_{j-1}, I_j, I_{j+1} be three consecutive contact intervals in 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 $e_j = \mathit{limits}(I_j) \cap \mathit{limits}(I_{j+1})$ (there is only one such edge, since $|Y| > 2$), and denote by $\mathcal{E}_j$ the polytope edge sequence intersected by e_j. Similarly, define $\mathcal{E}_{j-1}$ as the polytope edge sequence traversed by the transparent edge $e_{j-1} = \mathit{limits}(I_{j-1}) \cap \mathit{limits}(I_j)$. 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 Fig. 17.

We claim that $\bar{\mathcal{E}} = \mathcal{E}_{j-1} \cap \mathcal{E}_j$ is a contiguous subsequence of both sequences. Indeed, assume to the contrary that $\bar{\mathcal{E}}$ contains at least two subsequences $\bar{\mathcal{E}}_1, \bar{\mathcal{E}}_2$, and there is an edge $\bar{\chi}$ between them that belongs to only one of the sequences $\mathcal{E}_{j-1}, \mathcal{E}_j$. Then the region R of ∂P between the last edge of $\bar{\mathcal{E}}_1$, the first edge of $\bar{\mathcal{E}}_2$, e_{j-1} and e_j is contained in the region bounded by the contact intervals of Y and by their connecting transparent edges, and $\bar{\chi}$ must have an endpoint in R, contradicting the fact that this region does not contain any vertex of P. We can therefore use a binary

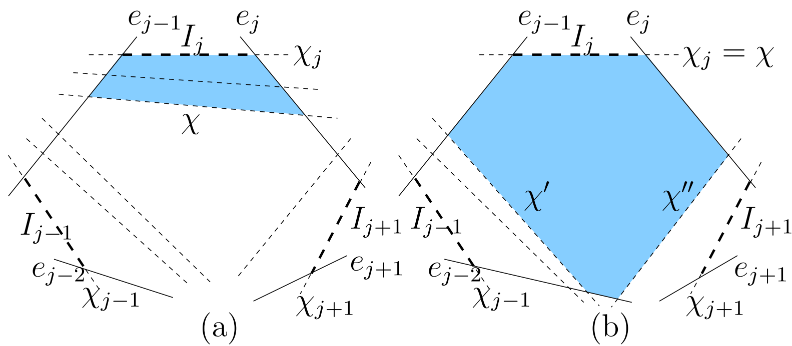

Fig. 17 There are two possible cases: (**a**) There is more than one edge in $\bar{\mathcal{E}}$, hence a building block of type III (whose unfolded image is shown *shaded*) can be extracted. (**b**) $|\bar{\mathcal{E}}| = 1$ (that is, $\chi_j = \chi$), therefore there must be a building block of type IV (whose image is shown shaded) that can be extracted

search to find the last polytope edge χ in $\bar{\mathcal{E}}$, by traversing the unfolding data structure tree T that contains $\mathcal{E}_{j-1}$ from the root r to the leaf that stores χ. To facilitate this search, we first search for ξ_j, which is the first edge of $\bar{\mathcal{E}}$. We then trace the search path $\mathcal{P}$ bottom-up. For each node μ on the path for which the path continues via its left child, we go to the right child ν, and test whether the edges stored at its leftmost leaf and rightmost leaf belong to the portion of $\mathcal{E}_j$ between χ_j and χ_{j+1}; for the sake of simplicity, we refer to this portion as $\mathcal{E}_j$. (As we will shortly argue, each of these tests can be performed in $O(1)$ time.) If both edges belong to $\mathcal{E}_j$, we continue up $\mathcal{P}$. If neither of them is in $\mathcal{E}_j$, then χ is stored at the rightmost leaf of the left child of μ. If only one of them (namely, the one at the leftmost leaf) is in $\mathcal{E}_j$, we go to ν, and start tracing a path from ν to the leaf that stores χ. At each step, we go to the left (resp., right) child if its rightmost leaf stores an edge that belongs (resp., does not belong) to $\mathcal{E}_j$.

To test, in $O(1)$ time, whether an edge $\chi^\star$ of P belongs to $\mathcal{E}_j$, we first recall that, by construction, all the edges of $\mathcal{E}_j$ intersect the original subface h_j of $S_{3\text{D}}$ from which e_j originates, and so they appear as a contiguous subsequence of the sequence of edges of P stored at the surface unfolding data structure at the appropriate x-, y-, or z-coordinate of h_j. Moreover, the slopes of the segments that connect them in the corresponding cross-section of P (which are the cross-sections of the connecting facets) are sorted in increasing order.

We thus test whether $\chi^\star$ intersects h_j. We then test whether the slope of the cross-section of the facet that precedes $\chi^\star$ lies within the range of slopes of the facets between the edges χ_j and χ_{j+1}. Clearly, $\chi^\star$ belongs to $\mathcal{E}_j$ if and only if both tests are positive. Since each of these tests takes $O(1)$ time, the claim follows. Hence, we can construct $\bar{\mathcal{E}}$ in $O(\log n)$ time.

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 that is the portion of χ bounded by e_j and e_{j-1}. See Fig. 17(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 Y and replace it by I'_j.

Otherwise, $\chi = \chi_j$. See Fig. 17(b). Denote by χ' (resp., χ'') the second edge in $\mathcal{E}_{j-1}$ (resp., $\mathcal{E}_j$); clearly, $\chi' \neq \chi''$. Since all blocks that contain either a vertex of P

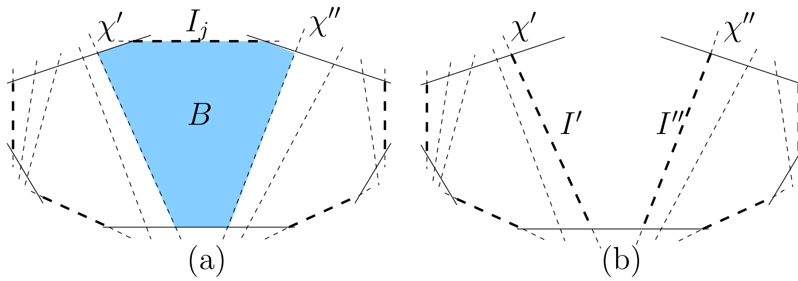

Fig. 18 (**a**) Before the extraction of B, Y contains five (*bold dashed*) contact intervals. (**b**) After the extraction of B, Y has been split into two new (cyclic) sequences Y', Y'' containing the respective contact intervals I', I''. I_j is no longer contained in any sequence in $\mathcal{Y}$

or a transparent edge endpoint are in $\mathcal{H}$, the edges χ_j, χ', χ'' bound a single facet, and there is a transparent edge that intersects both χ', 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', χ'' nearest to χ_j or, rather, nearest to e_{j-1} and to e_j, respectively (in Fig. 17(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 Y instead of I_j. If χ' contains I_{j-1} and χ'' contains I_{j+1}, 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 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 Y into two new cyclic subsequences, as shown in Fig. 18, and insert them into $\mathcal{Y}$ instead of Y.

In each iteration we compute the boundary of a single building block of type III or IV, hence there are $O(1)$ iterations; each performs $O(1)$ unfoldings, $O(1)$ binary searches, and $O(1)$ operations on constant-length lists, hence the time bound follows. □

3.2 Block Trees and Riemann Structures

In this section 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 that 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 three 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);

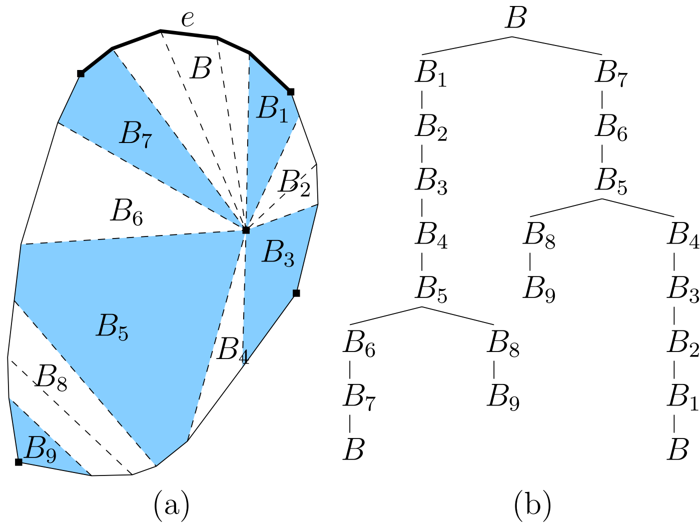

Fig. 19 (**a**) A surface cell c containing a single vertex of P and bounded by four transparent edges (*solid lines*) is partitioned in this example into ten building blocks (whose shadings alternate): B_1, B_3, B_7, B_9 are of type I, B, B_2, B_4, B_6 are of type II, B_8 of type III and B_5 of type 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

(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, which 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 $T_B(e)$—see Fig. 19.

Remark 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 Fig. 20 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.

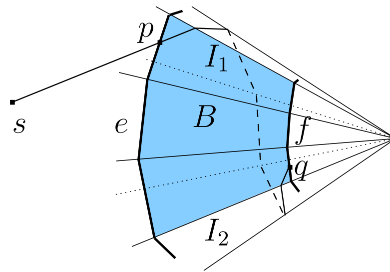

Fig. 20 The shortest path $\pi(s,q)$ enters the (*shaded*) building block B through the transparent edge e at the point p, leaves B through the contact interval I_1, and then reenters B through the contact interval I_2

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 Sect. 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). This structure is indeed similar to standard Riemann surfaces (see, e.g., [39]); 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 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).

Remark Concerning (i), note that without the Riemann structure, unfolding an arbitrary portion of ∂P may result in a self-overlapping planar region (making it difficult to apply the propagation algorithm)—see [11] for a discussion of this topic. However, there exist schemes of cutting a polytope along lines other than its edges that produce a non-overlapping unfolding—see [1, 6, 8, 36]. 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); however, related results [5, 30] do not suffice in our case, and we have not succeeded to prove this conjecture, which we leave for further research.

A *block sequence* $\mathcal{B} = (B_1, B_2, \ldots, B_k)$ is a sequence of building blocks of a surface cell c, so that for every pair of consecutive 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 $\mathcal{E}_1||(\chi_1)||\mathcal{E}_2||(\chi_2)||\cdots||(\chi_{k-1})||\mathcal{E}_k$, 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 3.13 *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. □

Note that Lemma 3.13 implies that each building block is stored in at most $O(1)$ nodes of $T_B(e)$.

Lemma 3.14 *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 Easy, and omitted. □

The following two lemmas summarize the discussion and justify the use of block trees. (Lemma 3.15 establishes rigorously the informal argument given right after the block tree definition.)

Lemma 3.15 *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, y are incident to two different respective contact intervals I_x, 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 3.16 *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 3.15). Hence, the elements of B form a path in $T_B(e)$ from the root node (which stores B) to a node that stores B', as asserted. □

Corollary 3.17 *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 Follows from Lemma 3.16. □

Lemma 3.18 (*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 3.13). There are two surface cells that bound e, and there are $O(1)$ building blocks of each surface cell, by Lemma 3.5. By Lemma 3.12, we can compute all the boundaries of all the building blocks in overall $O(n \log n)$ time. Hence the claim follows. □

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 3.17 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 3.18 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 Sects. 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*, 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. 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. In the analysis of the algorithm in Sects. 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 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 single 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.

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 [18]. Most of the problem-specific implementation details of the algorithm (which are quite different from those in [18]), as well as the final phase of the preprocessing for shortest path queries, are presented in Sect. 5.

The algorithm simulates a unit-speed *(true) wavefront W* expanding from s, and spreading along the surface of P. At *simulation time t*, W consists of points whose shortest path distance to s along ∂P is t. The true wavefront is a set of closed cycles; each cycle is a sequence of (folded) circular arcs (of equal radii), called *waves*. Each wave w_i of W 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 $\mathcal{E}_i$. The wave

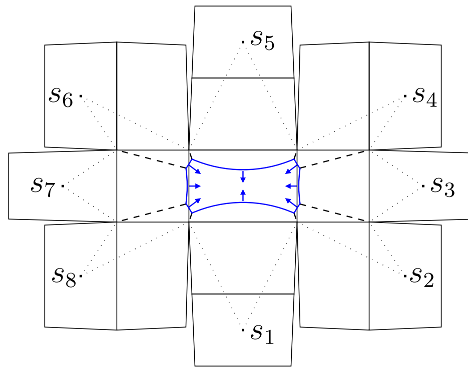

Fig. 21 The true wavefront W at some fixed time t, generated by eight source images $s_1, \dots, s_8$. The surface of the box P (see the 3D illustration in Fig. 22) is unfolded in this illustration onto the plane of the last facet that W reaches; note that some facets of P are unfolded in more than one 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

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 Fig. 21.

The wave w_i has at most two neighbors w_{i-1}, w_{i+1} in W, 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). As t increases and W 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; see Fig. 22. The bisector of the two consecutive generators s_i, s_{i+1} in W is denoted by $b(s_i, s_{i+1})$, and its unfolded image is a straight line.

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

(i) *Vertex event*, where W reaches either a vertex of P or some other boundary vertex (an endpoint of a transparent edge) of the Riemann structure through which W is propagated. As will be described in Sect. 5, the wave in W that reaches a vertex event splits into two new waves after the event—see Fig. 23. These are the only events when a new wave is added to W. Our algorithm detects and processes all vertex events.[7]

(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

[7] A split at a vertex of P is a "real" split, because the two new waves continue past v along two different edge sequences. A split at a transparent endpoint is an artificial split, used to facilitate the propagation procedure; see Sect. 5 for details.

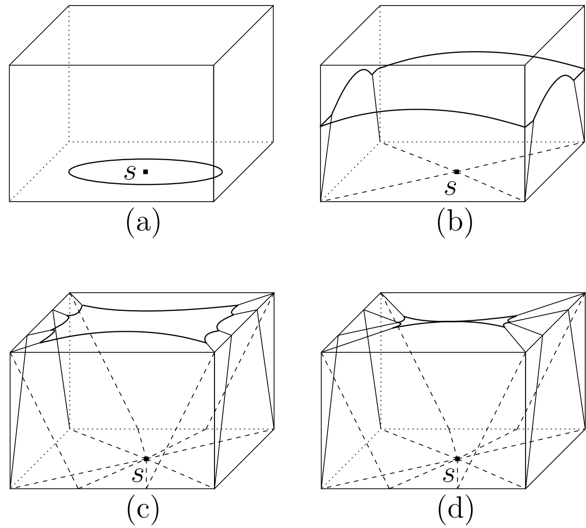

Fig. 22 W at different times t: (**a**) Before any critical event, it consists of a single wave. (**b**), (**c**) After the first four (resp., eight) vertex events W consists of four (resp., eight) (folded) waves. (**d**) After two additional critical events, which are bisector events, two waves are eliminated. Before the rest of the waves are eliminated, and immediately after (**d**), W disconnects into two distinct cycles

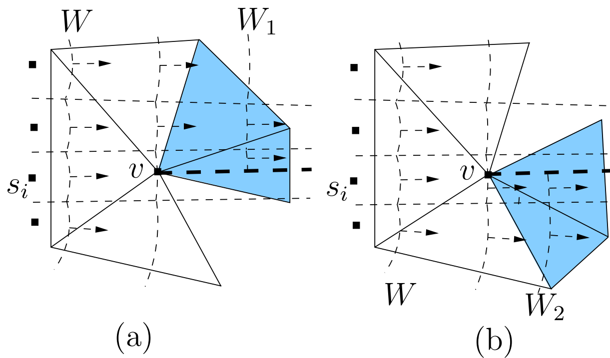

Fig. 23 Splitting the wavefront W at v (the triangles incident to v are unfoldings of its adjacent facets; note that the sum of all the facet angles at v is less than 2π). The *thick dashed line* coincides with the ray from s_i through v; it replaces the true bisector between the two new wavefronts W_1, W_2, which will later be calculated by the merging process. Each of 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**))

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 Sect. 4.3 for further details.

4.1 The Propagation Algorithm

One-Sided Wavefronts The wavefront propagates between transparent edges across the cells of the conforming surface subdivision 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

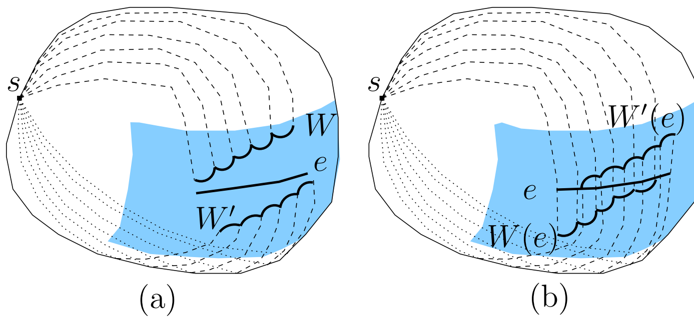

Fig. 24 (**a**) Two wavefronts W, W' are approaching e from two opposite directions, within $R(e)$ (*shaded*). (**b**) Two one-sided wavefronts $W(e)$, $W'(e)$, computed at the simulation time when e is completely covered by W, W', are propagated further within $R(e)$. However, some of the waves in $W(e)$, $W'(e)$ 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)

through e in opposite directions; together, these one-sided wavefronts carry all the information needed to compute the exact wavefront at e (but they also carry some superfluous information). 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 applies to the wavefront propagated by our algorithm.

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 that is the same plane in which we compute the unfolded image of e), so that: (1) There exists a pairwise openly disjoint decomposition of e into k nonempty intervals $e_1, \ldots, e_k$, appearing in this order along e, and (2) 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. The algorithm maintains the following crucial *true distance* invariant (see Fig. 24 for an illustration):

(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.

Remark For a fixed side of e, the corresponding one-sided wavefront $W(e)$ (implicitly) records the times at which the wavefront reaches the points of e from that side; note that $W(e)$ does not represent a fixed time t—each point on e is reached by the corresponding wave at a different time.

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. Sect. 2.3). To compute a one-sided wavefront at e, we propagate the one-sided wavefronts from each $f \in$ *input*(e) that has already been

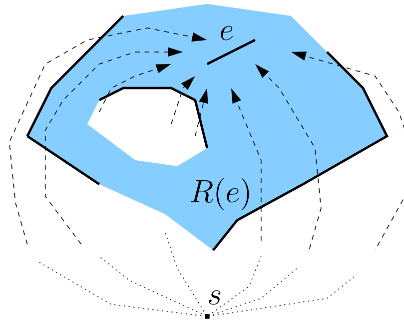

Fig. 25 The boundary of $R(e)$ (*shaded*) consists of two separate cycles. The transparent edge e and all the edges f in *input*(e) that have been covered by the wavefront before time *covertime*(e) are drawn as thick lines. The wavefronts $W(f,e)$ that contribute to the one-sided wavefronts at e have 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)

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 Fig. 25 for an illustration. The algorithm propagates the wavefronts inside $O(1)$ unfolded images of (portions of) $R(e)$, using the Riemann structure defined in Sect. 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 $f \in$ *input*(e) to the image of e, must not bend (cf. Sect. 2.1 and Sect. 3). Because the image of the appropriate portion of $R(e)$ is not necessarily convex, its reflex corners may block portions of wavefronts from some edges of *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 *input*(e). For any point $p \in e$, the shortest path from s to p passes through some $f \in$ *input*(e) (unless $s \in R(e)$), so constraining the source wavefronts to reach e directly from an edge in *input*(e), without leaving $R(e)$, does not lose any essential information.

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

Lemma 4.1 *For any transparent edge* e, *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 $input(f)$ 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), and $|$*input*$(f)| = O(1)$, by construction. □

Remark As a wavefront is propagated from an edge $f \in$ *input*(e) to e, it may cross other intermediate transparent edges g (see Fig. 26). Such an edge g will be processed at an interleaving step, when wavefronts from edges $h \in$ *input*(g) are propagated to g

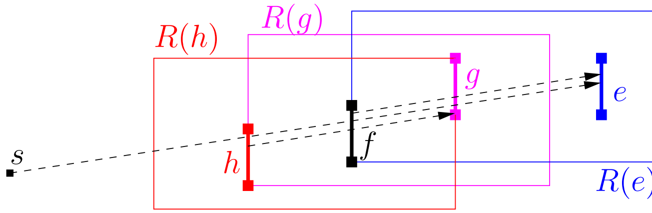

Fig. 26 Interleaving of the well-covering regions. The wavefront propagation from $h \subset \partial R(g)$ to g passes through f, and the propagation from $f \subset \partial R(e)$ to e passes through g

(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.

The Simulation Clock The simulation of the wavefront propagation is loosely synchronized with the real "propagation clock" (which 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.

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 $d_S(s,e) \le \tilde{d}_S(s,e) < d_S(s,e) + |e|$. 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 Sect. 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 the *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:

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. 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 $f \in input(e)$ (these wavefronts are propagated 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 Sect. 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 Sect. 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 a constant number of 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 (which reaches e at the time $|\pi'| = t_e = \tilde{d}_S(s, e)$) crosses some $f \in input(e)$ at an earlier time t_f, where $d_S(s, f) \leq t_f < \tilde{d}_S(s, f) + |f|$; we may assume that f is the last such edge of $input(e)$. Note that we must have $t_e \geq t_f + d_S(e, f)$. By (W3$_S$), $d_S(e, f) \geq 2|f|$, and so $t_e \geq d_S(s, f) + 2|f|$. Since $\tilde{d}_S(s, f) < d_S(s, f) + |f|$, we have

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

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 $e \in output(f)$ (i.e., $f \in input(e)$), which consists of $O(1)$ edges, by construction.

(c) Any path π that is part of a one-sided wavefront at e must satisfy $d_S(s, e) \leq |\pi| < \tilde{d}_S(s, e) + |e|$ (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 $f \in input(e)$, 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 $\tilde{d}_S(s, f) + |f| < \tilde{d}_S(s, e) + |e|$. □

Remark The synchronization mechanism above 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 all the wavefronts that reach e, 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 another wavefront.[8]

Topologically Constrained Wavefronts Let f, e be two transparent edges so that $f \in input(e)$, and let H be a homotopy class of simple geodesic paths connecting f to e within $R(e)$ (recall that there might be multiple homotopy classes of that kind; see Sect. 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

[8]For a detailed discussion of why we use the bound $2|e|$ rather than just $|e|$ see the description of the simulation time maintenance in Sect. 5.3.1.

to e that belong to H. In Sect. 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.) We say that $W_H(f, e)$ is a *topologically constrained wavefront* (by H). To simplify notation, we omit H whenever possible, and simply denote the wavefront, somewhat ambiguously, as $W(f, e)$.

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, which is the locus of all points traversed by all (geodesic) paths in H (see Fig. 27). 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 [14] for a similar analysis).

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 Fig. 27. 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 Fig. 27 for an example.[9]

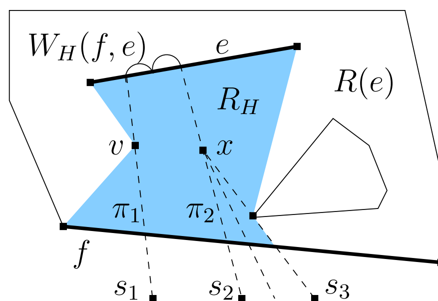

Fig. 27 The "hourglass" region R_H that is traversed by all paths in H is *shaded*. The extreme artificial bisectors of the topologically constrained wavefront $W_H(f, e)$ are the paths π_1 (from the extreme generator s_1 through the vertex v of P, which is one of the constraints of H) and π_2 (from the generator s_2, which became extreme when its neighbor s_3 was eliminated at a bisector event x, through the location of x)

[9]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.

4.2 Merging Wavefronts

Consider the computation of the one-sided wavefront $W(e)$ at a transparent edge 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 $f \in input(e)$, that contain waves that reach e from the right (not later than at time $covertime(e)$). 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 opposite one-sided wavefront at e 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 $input(e)$.

We call the set of all points of e claimed by a contributing wavefront $W(f,e)$ the *claimed portion* or the *claim of* $W(f,e)$. The following lemma implies that this set is a (possibly empty) *connected* subinterval of e.

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 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 Fig. 28(a). By (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. 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

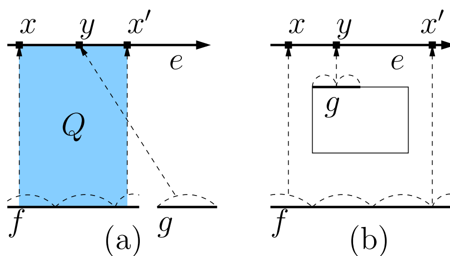

Fig. 28 (**a**) $W(g,e)$ cannot claim the point y, for otherwise the shortest path $\pi(s,y)$ (which crosses the transparent edge g) would have to cross one of the paths $\pi(s,x), \pi(s,x')$, which is impossible for shortest paths. The region Q delimited by f, e, and the portions of $\pi(s,x), \pi(s,x')$ 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

and e. 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. □

Remark Lemma 4.3 may fail if $W(f, e)$ is *not* a topologically constrained wavefront; see Fig. 28(b) for an example. 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.

We now proceed to describe the *merging process*, applied to the contributing wavefronts that reach a transparent edge e from a fixed side; the process results in the construction of the corresponding one-sided wavefront at e. Most of the low-level details of the process are embedded in the procedures supported by the data structure described in Sect. 5.1; for now, before proceeding with Lemma 4.4, we briefly review the basic operations, and assert their time complexity bounds. Each contributing wavefront W is maintained as a list of generators in a balanced tree data structure; 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 Sect. 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 its claim 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 results in the algorithm asserted in the lemma.

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 $W(f, e)$, $W(g, e)$ 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. Using the surface unfolding data structure of

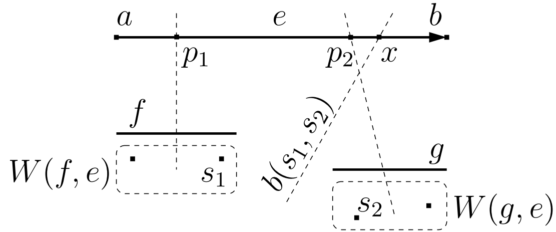

Fig. 29 The source image s_2 is eliminated from $W(e)$, because its contribution to $W(e)$ must be to the left of p_2 and to the right of x, and therefore does not exist along e

Sect. 2.4, we can compute the polytope edge sequence $\mathcal{E}_e$ crossed by e, in $O(\log n)$ time, and compare $U_{\mathcal{E}_e}(x)$ with $U_{\mathcal{E}_e}(y)$.

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 $W(f,e)$.

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 Fig. 29. 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 $W(f,e)$.

Consider next the time complexity of this process. Merging each of the $O(1)$ pairs $W(f,e)$, $W(g,e)$ of wavefronts involves $O(1+k)$ operations, where k is the number of generators that are deleted from the wavefronts during that merge, 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, each of these operations can be implemented in $O(\log n)$ time. Summing over all $O(1)$ pairs $W(f,e)$, $W(g,e)$, the bound follows. □

The following lemma proves the correctness of the process, with the assumption that the propagation procedure, whose details are not provided yet, is correct.

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 $f \in \mathit{input}(e)$. (ii) The one-sided wavefronts at each edge $f \in \mathit{input}(e)$ 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 established 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 Sect. 5.2, and when we merge the wavefronts that reach the same transparent edge, as described in Sect. 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 a transparent edge e to another edge g to compute the wavefront $W(e, g)$, where $g \in \mathit{output}(e)$. 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 (as we show in Lemma 5.6, this is the only kind of events that occur when waves from the *same* topologically constrained wavefront collide with each other); see Fig. 30(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.

A bisector event, at which the *first* generator s_1 in the propagated wavefront is eliminated, is treated somewhat differently; see Fig. 30(b), (c) for an illustration. In

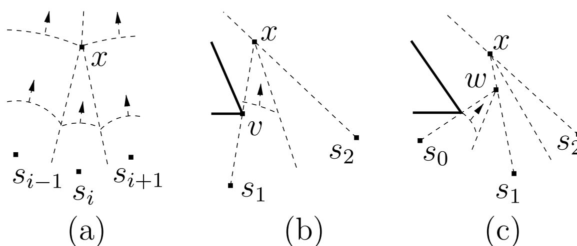

Fig. 30 When a bisector event (of the first kind) takes place at x: (**a**) The wave of s_i is eliminated, and the new bisector $b(s_{i-1}, s_{i+1})$ is computed. (**b**), (**c**) The wave of s_1 is eliminated, and the ray from s_2 through x becomes the leftmost (artificial) bisector of W, instead of the former leftmost bisector, which is the ray from s_1 through either (**b**) a transparent edge endpoint v (a visibility constraint), or (**c**) the location w of an earlier bisector event, where s_0, the previous leftmost generator of W, has been eliminated

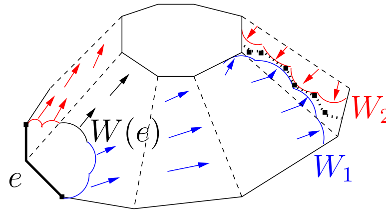

Fig. 31 $W(e)$, propagated from e, is split inside $R(e)$ when it reaches the inner (*top*) boundary cycle. Then the two new topologically constrained wavefronts partially collide into each other, creating a sequence of bisectors (*dotted lines*, bounded by *thick points* where bisector events of the second kind occur), eliminating a sequence of waves in each wavefront

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 with each other. Our algorithm does not explicitly detect these events; however, they are all (implicitly) considered at the query processing time, as described in Sect. 5.4, and some of them undergo additional (albeit still implicit) processing, as briefly described next.

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 $W(f, g)$ that also reaches g, which eliminates the wave of s_i. This event is implicitly recognized by the algorithm when s_i is deleted from $W(e, g)$ during the merging process at g.

Another kind of such an event occurs when a one-sided wavefront $W(e)$ is split during its propagation inside $R(e)$ (either at of a vertex of P or at 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 Fig. 31. The algorithm implicitly processes some of these events, by realizing that these waves attempt to exit the current block tree, by re-entering an already visited building block. The algorithm then simply discards these waves from further processing; see Sect. 5.3.1.

Tentatively False and True Bisector Events Consider the time $t = \mathit{covertime}(e)$. 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 that could have taken place before time t. As described in Sect. 5, the algorithm detects these (currently considered as) "false" bisector events when the wavefronts from the edges in $\mathit{input}(e)$ are propagated to e, but the generators that are eliminated in these events are not deleted from their corresponding

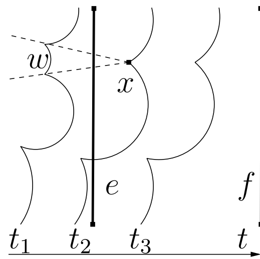

Fig. 32 The bisector event at x occurs at time t_2. It is first detected when the wavefront is propagated toward the transparent edge e, which has not been fully covered yet. Since x is beyond e, the event is currently considered false (and the eliminated wave w is not deleted from the wavefront, so 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

contributing wavefronts before time t. This is done to ensure that the invariant (TD) is satisfied. However, such a bisector event 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 Fig. 32. Further details are given in Sect. 5, where we also show that the number of all "true" and "false" processed events is only $O(n)$.

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. To find the true claimer of x (or any other query point), we make use of the fact that x belongs to only $O(1)$ well-covering regions, each of which is traversed by only $O(1)$ wavefronts; knowing the claimer of x in each of these wavefronts gives us the "global" claimer of x—see Sect. 5.4.

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 [18]):

1. *List operations*: CONCATENATE, SPLIT, and DELETE.[10] Each operation is applied to the list of generators that represents the wavefront at any particular simulation time.

[10]Note that the algorithm does not use INSERT operations; a new wave is created only during a SPLIT operation, and generating it is part of the SPLIT. Similarly, the omitted CREATE operation is performed only once, when the first singleton wavefront at s is created.

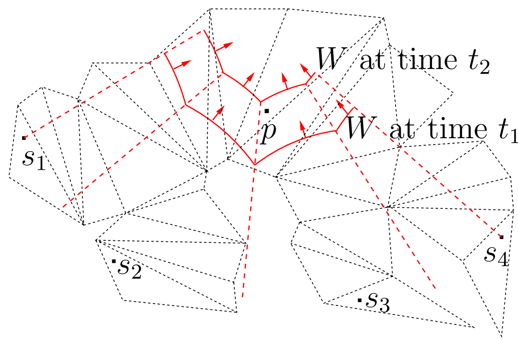

Fig. 33 The wavefront W at simulation times t_1 and t_2 consists of four source images $s_1, \dots, s_4$, all unfolded to one plane at time t_1 and to another plane at time t_2 (for this illustration, both planes are the same—this is the plane of the facet that contains the point p). In order to determine 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

2. *Priority queue operations*: We assign to each generator a priority (as defined below in Sect. 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*: (a) To compute explicitly each source image s_i in the wavefront at time t, we need to unfold the maximal polytope edge sequence of s_i at t—this operation is referred to as an "unfolding query"; the unfolding structure needs to be updated as the wavefront advances. (b) 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 a query point q. That is, we SEARCH in the generator list for a claimer of q (without considering other wavefronts or possible visibility constraints); see Fig. 33, and see later for more precise details.

All these types of operations can be supported by a data structure based on balanced binary search trees, with the generators stored at the leaves [15]. In particular, the "bare" list operations (ignoring the maintenance of priorities and unfolding data) take $O(\log n)$ time each, using standard machinery [15, 37]. Moreover, one can also update the extra unfolding fields (described in the following paragraphs) as these list operations are executed (so that the operations retain their $O(\log n)$ time). Although not completely straightforward, the manipulation of the unfolding fields is still simple enough, so that we omit it here—we present the full details in [34]. The 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; the actual implementation details are fairly standard, 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 $v_1 = \text{root}, v_2, \ldots, v_k =$ 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 [32, 34]), so composition of any pair of transformations takes $O(1)$ time. 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.

For each internal node v, let ($v = v_1, v_2, \ldots, v_k =$ the rightmost leaf of the left subtree of v) be 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) be the path from v to $v'_{k'}$. To perform the SEARCH operation efficiently, we store at v the *bisector image* $b[v] = b(U[v_1]U[v_2]\cdots U[v_k](s), U[v'_1]U[v'_2]\cdots U[v'_{k'}](s))$, which is the bisector between the source image stored at v_k and the source image stored at $v'_{k'}$, 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). During any operation that modifies the data structure, we always maintain the invariant that $b[v]$ is unfolded onto $\Lambda(v)$. As already said, the updating of the fields $U[v], b[v]$, at nodes v affected by tree rebalancing rotations, is quite simple, and described in [34].

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.

Note that 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).

Typical Manipulation of the Structure 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 Sect. 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 $U[r']$

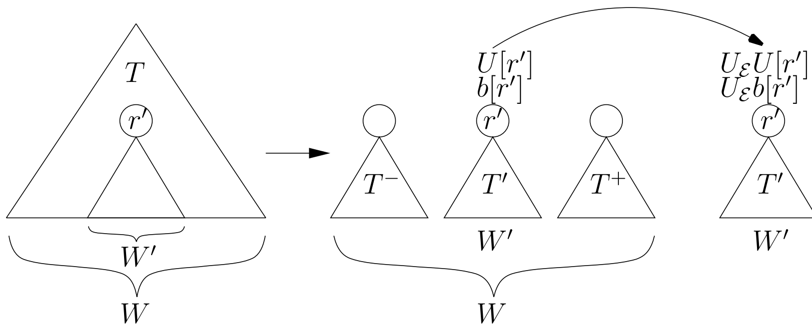

Fig. 34 T is split into three subtrees T^-, T', T^+, where T' stores the sub-wavefront W' of W. Then the unfolding fields stored at the root r' of T' are updated

by $U_{\mathcal{E}}U[r']$ and $b[r']$ by $U_{\mathcal{E}}b[r']$; see Fig. 34. Finally, we concatenate T^-, the new T', and T^+, into a common new tree T.

Remark 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 [28], which stores the incidence information between m nonintersecting geodesic paths and n polytope edges; 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 the worst case.)

Maintaining all Versions We also require our data structure to be *confluently persistent* [12]; 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 [12]) 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.

We also use the persistence of the data structure to implement the wavefront propagation through a block tree, as described in Sect. 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.

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 [20]. 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. 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 Sect. 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 Sect. 4.2, which, together with the data structure details presented in Sect. 5.1, implies that all the merging procedures can be executed in $O(n \log n)$ time.

Let e be a transparent edge. 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 that correspond to block sequences from e to g within $R(g)$ (see Sect. 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 $W_H(e, g)$.[11] 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, 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 W); during the propagation, these initial wavefronts are further split into a total of k sub-wavefronts, one per homotopy class.

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 $g \subset \partial c$. To compute $W(e, g)$ for all transparent edges $g \in output(e)$, possibly not belonging to ∂c, we proceed as follows. We propagate $W(e)$ cell-by-cell inside $R(g)$ from e to g, and 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

[11] The initial singleton wavefront $W(s)$ from s to a transparent edge g on the boundary of the cell that contains s is propagated similarly.

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.

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 Sect. 4.1.

We propagate the wavefront separately in each of the $O(1)$ block trees of the Riemann structure $\mathcal{T}(e)$. 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 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 3.17, these component wavefronts collectively encode all the shortest paths from s to points p of c that 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, $W = W(t_0) = W(e_B)$. Note that even though we need to start the propagation from e at simulation time $t_0 = covertime(e)$, the actual starting time may be strictly smaller, since there may have been bisector events beyond e that have occurred before time $covertime(e)$. In

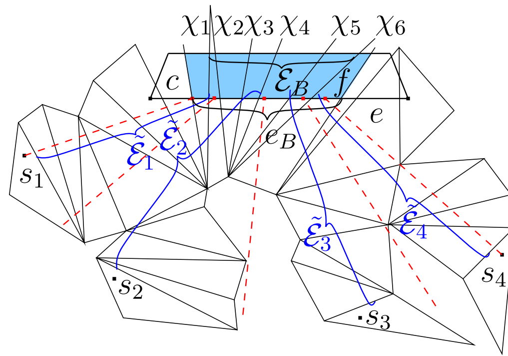

Fig. 35 The block B is *shaded*; the edge sequence associated with B is $\mathcal{E}_B = (\chi_1, \ldots, \chi_6)$. $W(e_B)$ consists of four source images $s_1, \ldots, s_4$, all unfolded to the plane of the facet f before the simulation of the propagation into $T_B(e)$ starts (that is, the last facet 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 $\mathcal{E}_4 = \tilde{\mathcal{E}}_4 \setminus (\chi_6)$

this case, these events need now to be processed (up to now, they have been detected by the algorithm but not processed yet), and we set t_0 to be the time when the earliest among them takes place.

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 $\mathcal{E}_B^{\text{rev}}$. In the former case we extend $\tilde{\mathcal{E}}_i$ by the appropriate suffix of $\mathcal{E}_B$ (which takes us to f in Fig. 35). In the latter case we truncate $\tilde{\mathcal{E}}_i$ at the first polytope edge of $\mathcal{E}_B^{\text{rev}}$ 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); it only stores and composes their unfolding transformations, as described in Sect. 5.1. Denote by $\Lambda(W)$ the (common) destination plane of all the $U_{\mathcal{E}_i}$. We do not alter $\Lambda(W)$ until the propagation of W in $T_B(e)$ is completed (and then $\Lambda(W)$ is updated, as described below). That is, as we traverse new blocks of $T_B(e)$, we unfold them all to the plane $\Lambda(W)$. 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 $\mathcal{E}$ of s is empty, and $U_{\mathcal{E}}$ 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 *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 $B \cap e$. Since $T_B(e)$ has $O(1)$ nodes, and each block has $O(1)$ boundary elements, $\mathcal{C}$ contains only $O(1)$ elements; see Fig. 36.

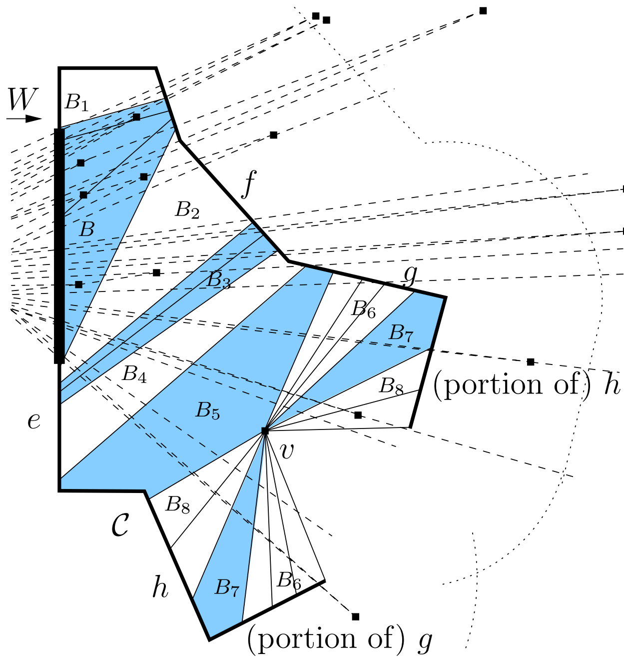

Fig. 36 Bisector events (the *thick square points*), some of which are processed during the propagation of the wavefront W from the transparent edge portion e_B (the *thickest segment* in this figure) through the building blocks (their shadings alternate) of the block tree $T_B(e)$. The unfolded transparent 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 $W(e_B, f)$

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 $\mathcal{C}$. (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, as illustrated in Fig. 36.) 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 subset of segments 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 processing of these events, we provide here the intuition behind the (somewhat unorthodox implementation of the) low-level procedures.

The purpose of the propagation of W in $T_B(e)$ is to compute 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 with respect to the propagation of W in $T_B(e)$*; that is, events that take place in $T_B(e)$ that would have been vertices of SPM(s) if there were no other wavefronts except W. For the sake of brevity, in the rest of this section we refer to these events simply as *true events*. Unfortunately, it is difficult to determine in "real time" 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 take place later than some earlier true event that has not yet been detected (and processed).

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' intersect beyond e_B, either in B or beyond it. 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 Fig. 37(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, as illustrated in Fig. 37(c).

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 that 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, 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

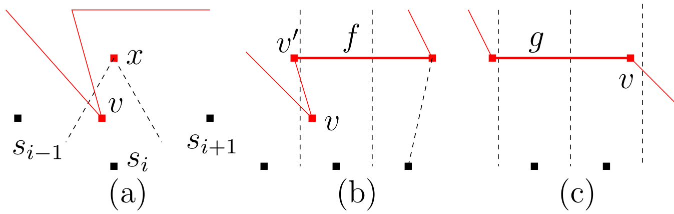

Fig. 37 An earlier vertex event at $v \in \mathcal{C}$ can be detected later: (**a**) while processing a false bisector event x; (**b**) while processing a vertex event at an endpoint v' of a segment $f \subset \mathcal{C}$, when f is ascertained to be covered by W; (**c**) when the segment $g \subset \mathcal{C}$, incident to v, is ascertained to be covered by W

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).

Path Tracing Let $x \in \Lambda(W)$ be an (unfolded) image of some point of c, and let $s' \in W$ be a source image. To determine whether the path to x from s' does or does not meet $\mathcal{C}$, and, in the former case, to also determine the first intersection point (along the path $\pi(s', x)$) with $\mathcal{C}$, we *trace* $\pi(s', x)$ either up to x, or until it intersects $\mathcal{C}$—whichever occurs first—as follows.[12]

The tracing is done by following the sequence of blocks traversed by $\pi(s', x)$, which forms a path in $T_B(e)$. At each block B' that we encounter, we test whether $\pi(s', x)$ terminates within B', and, if not, we find the edge of $\partial B'$ through which $\pi(s', x)$ leaves B'. If we reach x, or if the exit edge of $\partial B'$ is a portion of $\mathcal{C}$, we stop the tracing. Otherwise we exit B' through a contact interval I, and proceed to the next block beyond I. (It is also possible that we reach a contact interval I which is a "dead-end" in $T_B(e)$, and is thus a portion of $\mathcal{C}$.)

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 Sects. 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 Follows from the description of the tracing procedure. □

Note that we can similarly trace any path π of W until it intersects $\mathcal{C}$, without specifying any terminal point on π, as long as the starting direction of π in $\Lambda(W)$ is well defined.

[12] Here and 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).

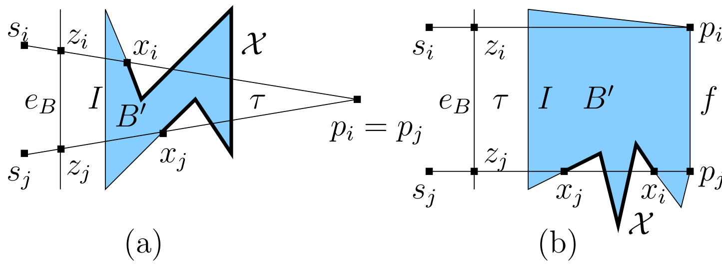

Fig. 38 (**a**) $\pi(s_i, p_i), \pi(s_j, p_j)$ leave B' through two different contact intervals of $\partial B'$. Here $p_i = p_j$, and τ is the triangle $z_i p_i z_j$. (**b**) $\pi(s_i, p_i)$ reaches $p_i \in B'$ and $\pi(s_j, p_j)$ leaves B' at the point x_j. Here $p_i \neq p_j$, and τ is the quadrilateral $z_i p_i p_j z_j$. The portion $\mathcal{X}$ of $\partial B'$ is highlighted in both cases

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 Fig. 38.

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 $\Lambda(W)$, 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 Fig. 38(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, starting from 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 $\overline{z_1 z_2}$.

Consider next the case in which 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 Fig. 38(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. □

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.[13] When the time $t_{\text{stop}}(W)$ is reached, the following holds: Either $t_{\text{stop}}(W) = +\infty$ (see Fig. 39(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 Fig. 39(b)), and there exists some sub-wavefront $W' \subseteq W$ that 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.

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

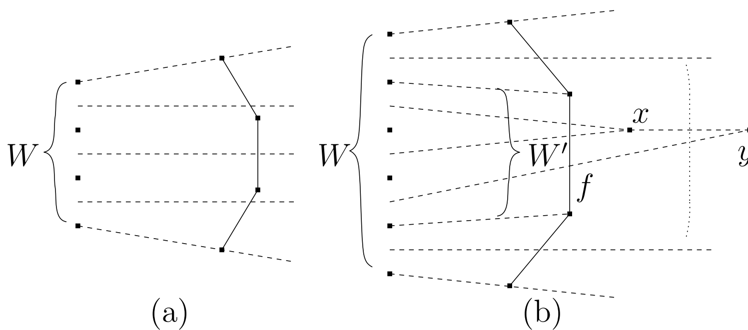

Fig. 39 (**a**) The stopping time $t_{\text{stop}}(W) = +\infty$. (**b**) The stopping time $t_{\text{stop}}(W') = t_{\text{stop}}(f) < +\infty$; the *dotted line* indicates the stopping time (or distance) at which we stop processing bisector events: the event at x has been processed before $t_{\text{stop}}(W')$, while the event at y has been detected but not processed

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

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 $t_{\text{stop}}(W) = +\infty$.

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).

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 Fig. 40(a). 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 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 Fig. 40(b).

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 Fig. 36). 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 3.17; therefore

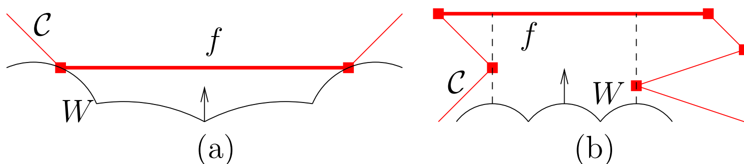

Fig. 40 (**a**) Both endpoints of f are reached by W before f is covered by W. (**b**) W actually reaches only a portion of f (between the two *dashed lines*), because of visibility constraints

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 $t_{\mathrm{stop}}(W)$, 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 Sect. 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 $\mathit{priority}(s_i)$.

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 Fig. 41(a)), which means that neither π_1 nor π_2 intersects $\mathcal{C}$, 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; see Fig. 41(b).) 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 $\mathit{priority}(s_{i-1}) := +\infty$; otherwise $\mathit{priority}(s_{i-1})$ 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. 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 b of W as a ray from s_2 through x; the priority of s_2 is recomputed as above, intersecting b with the other bisector of s_2. 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 $T_B(e)$*: 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 Fig. 41(b–d) for an illustration.

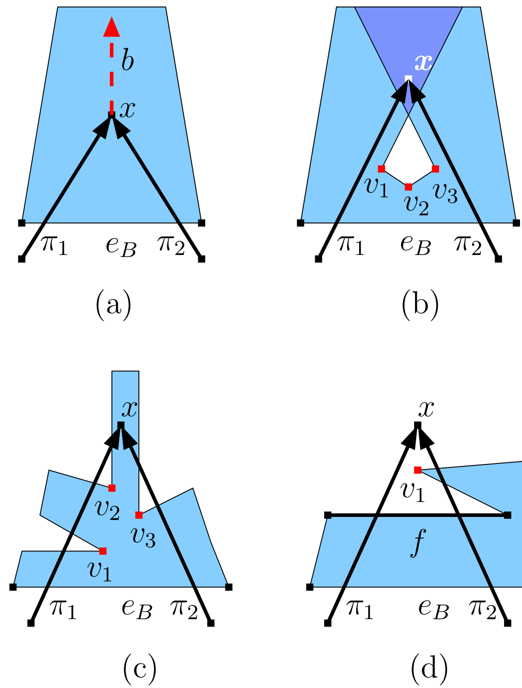

Fig. 41 In (**a**) x is a true bisector event; the new bisector b between the generators of π_1, π_2 is shown dashed. In (**b–d**) x is a false candidate. (**b**) π_1, π_2 do not intersect $\mathcal{C}$, but reach x through different layers of the Riemann structure that overlap each other. At least one 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 $\mathcal{C}$ (before reaching x). (**d**) $\mathcal{C}(\pi_1) = \mathcal{C}(\pi_2) = f$. No vertex of $\mathcal{V}$ (here $\mathcal{V} = \{v_1\}$) is visible from the portion of e_B between π_1 and π_2

If π_1 intersects $\mathcal{C}$, denote the first such intersection point (along π_1) by z and the segment $\mathcal{C}(\pi_1)$, which contains z, by 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. 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 $t_{\text{stop}}(W) = t_{\text{stop}}(f) = d_z + |f| \le t = d_z + d(z, x)$, where $d(z, x)$ is the distance from z to x along π_1; see Fig. 42. Hence $d(z, x) \ge |f|$. 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 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 $\mathcal{V}$.

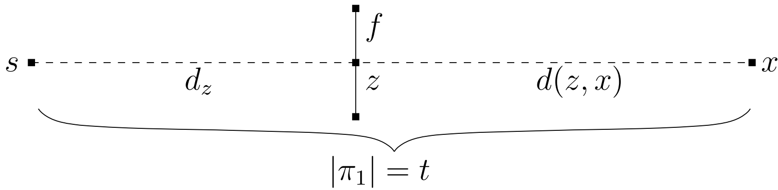

Fig. 42 If $d_z + |f| \le t = d_z + d(z, x)$, then $d(z, x) \ge |f|$

Assume first that π_1, π_2 satisfy the assumptions of Lemma 5.3; it follows that $\mathcal{V}$ is not empty (see Fig. 41(b), (c)). We trace the path from each generator in W' to each vertex v of $\mathcal{V}$, and compute $claimer(v)$ (which satisfies $d(claimer(v), v) = \min\{\{d(s', v) \mid v \text{ is visible from } s' \in W'\} \cup \{+\infty\}\}$). Denote by u the vertex of $\mathcal{V}$ so that $t_u := d(claimer(u), u) = \min_{v\in\mathcal{V}} d(claimer(v), v)$; 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 $t_x = priority(s_i)$ is the current simulation time). 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 $s' := claimer(u)$ 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 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 Sect. 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 Fig. 41(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 should not be 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 $priority(s_i) := +\infty$ (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 5.9 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 $priority(s_i) := +\infty$. 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 $W(e, f)$ 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, and keep a list of pointers to all the currently marked generators; when their priorities must be reset, we go over the list, fixing each generator and removing it from the list).

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 5.9 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 at least $t_{\text{stop}}(W)$, 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 $T_B(e)$ at simulation 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 which occur in c and 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 we process the (only relevant) segment f in the former case; and so, we consider only 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 5.9 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 a 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 point of f closest to a which is intersected by an extreme bisector of W—see Fig. 43(a). (If there is no such a', we can already determine that W claims no points on f, and no further processing of f is needed—see Fig. 43(c).) 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 $\pi(\mathit{claimer}(a), a)$ (resp., $\pi(\mathit{claimer}(b), b)$); otherwise denote by π_1 (resp., π_2) the path $\pi(\mathit{claimer}(a'), a')$ (resp., $\pi(\mathit{claimer}(b'), b')$). Denote by W' the sub-wavefront of W between the generators of π_1 and π_2 (inclusive), and use

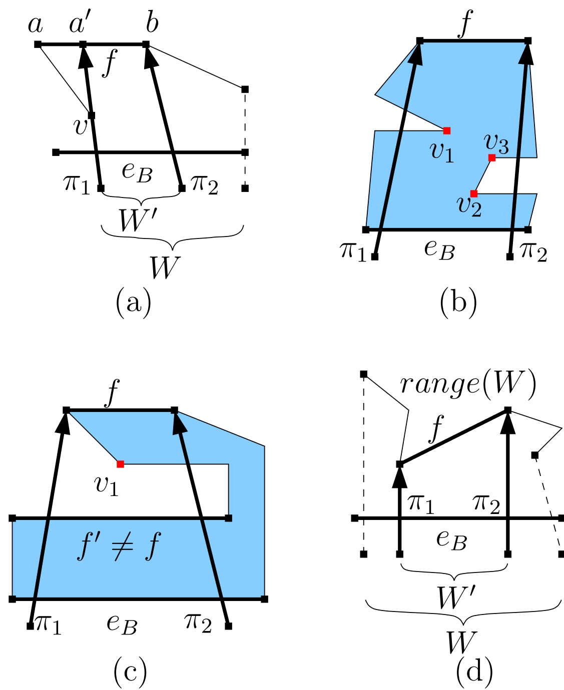

Fig. 43 Processing a covered segment f of *range*(W). (**a**) The endpoint a of f is not claimed by W, and π_1 is the shortest path to the point a' closest to a and claimed by W; the generator of π_1 is extreme in W (which has already been split at v). (**b**) At least one vertex of $\mathcal{V} = \{v_1, v_2, v_3\}$ (namely, v_2) is visible from the entire portion of e_B between π_1 and π_2. (**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 $d_f = |\pi_1| < |\pi_2|$, W is first split at the generator of π_1

π_1, π_2 to define (and compute) $\mathcal{V}$ as in the processing of a candidate bisector event (described above).

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 its claimer in W' (see, e.g., Fig. 43(b)). Then the case is processed as Case (ii) of a candidate bisector event, 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 paths $\pi(\mathit{claimer}(v), v)$. (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 Fig. 43(c))—we *ignore* f in the rest of the present simulation, and update $t_{\mathrm{stop}}(W) := \min\{t_{\mathrm{stop}}(f') | f' \in \mathit{range}(W) \setminus \{f\}\}$. 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 Fig. 43(d)). We restart the 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 (not *ignored*) 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 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 Sect. 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.

To summarize, we trace $O(1)$ paths and perform at most $O(1)$ SPLIT and SEARCH operations, for each of $O(1)$ segments of $\mathcal{C}$. Then we perform at most one source unfolding data 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 that 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.

Correctness and Complexity Analysis We start by observing, in the following lemma, a basic property of W that asserts that distances from generators *increase* along their bisectors.

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) *the distances from s_i and s_j to points y on the portion of $b(s_i, s_j)$ that is not claimed by s', increase as y moves away from x.*

Proof In the plane $\Lambda(W)$, consider the Voronoi diagram of the three sites s_i, s', s_j, whose sole vertex is x. The line containing e intersects exactly two Voronoi edges, because it meets the Voronoi cells $V(s_i)$, $V(s')$, $V(s_j)$ of all the three sites. Moreover, by assumption, $e \cap V(s')$ lies between $e \cap V(s_i)$ and $e \cap V(s_j)$. Hence, the Voronoi edge that e misses is between $V(s_i)$ and $V(s_j)$, implying that $b(s_i, s_j)$, between e and x, is fully contained in $V(s')$, as asserted—see Fig. 44(a). The same argument also implies (ii). □

Lemma 5.5 *Assume that all bisector events of W that have occurred up to some time t have been correctly processed, and that 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 $d(s_i, p) \le t$, meaning that the claim is only with respect to the current generators in W (at time t), and that we ignore any visibility constraints of $\mathcal{C}$. Denote by $R(s_i)$*

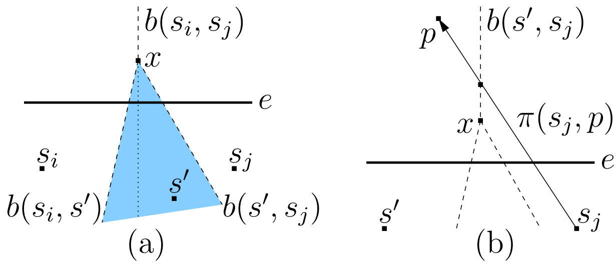

Fig. 44 (**a**) The bisector $b(s_i, s_j)$, between e and x, must be fully contained in $V(s')$ (*shaded*). (**b**) If the bisector $b(s', s_j)$ is already computed in the wavefront W, then the path $\pi(s_j, p)$, which intersects $b(s', s_j)$, cannot be encoded in W

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; hence, p is enclosed between them.

For the second claim, assume to the contrary that there exists a generator $s_j \neq s_i$ in W so that $p \in R(s_j)$ too. Denote by q the first point along $\pi(s_j, p)$ that is equally close 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 part of $W(e_B)$; therefore $b(s', s_j)$ emanates from a bisector event location x that lies within c—see Fig. 44(b). By Lemma 5.4, $d(s', x) < d(s', q) < d(s', p) \leq t$; hence, the bisector event when $b(s', s_j)$ is computed occurs before time t, and therefore, by assumption, $b(s', s_j)$ 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 when 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 5.9 below.

Lemma 5.6 *Assume that all bisector events of W that have occurred up to some time t have been correctly processed, and that 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 simulation 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, $d(s_i, x) = d(s_j, x) = t \leq d(s_k, x)$ 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 Fig. 45 for an illustration.

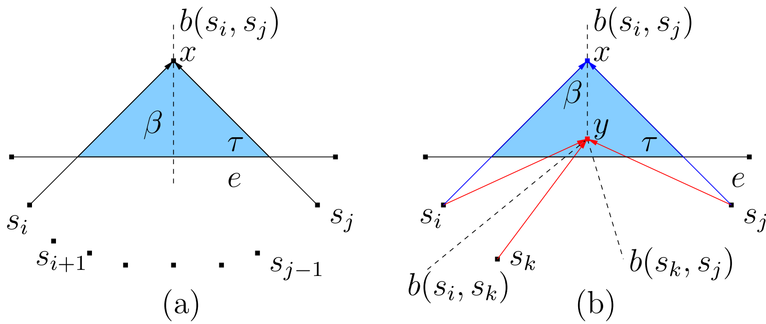

Fig. 45 The waves from the source images s_i, s_j collide at x. Each of the two following cases contradicts the assumption in the proof of Lemma 5.6: (**a**) The portion β of $b(s_i, s_j)$ intersects the transparent edge e; (**b**) The generator s_k is eliminated at time $t_y = d(s_i, y) < d(s_i, x) = t$

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 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 Fig. 45(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 y within τ—see Fig. 45(b). Clearly, y is the location of a bisector event in which some generator $s_k \in W$ is eliminated at time $t_y = d(s_i, y) = d(s_j, y)$. 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 generators of W that are consecutive at the time the bisector event occurs. 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 topologically constrained wavefronts, $\partial R(W', t)$ consists only of e_B and of the unfolded images of the waves and of the extreme bisectors of W' at time t. 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 that are not covered by W' at time t).*

Proof Consider the first time at which $R(W', t)$ becomes punctured. When this happens, $R(W', t)$ must 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 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 x, or are adjacent in the wavefront that covers the segment f. 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. □

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, $t_1 < \cdots < t_m$.

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 $t \le t''$ and has not been detected prior to 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 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 x'' that is still undetected. In this case, we create a new version of W at time t', but it will later be declared invalid, when we finally detect x''.

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, the generator that is eliminated at x' is marked in it, 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'.

Assume now that at the simulation time t_k (for $1 \le k \le m$) 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 does not encode events 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 5.9 *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. 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 that 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.

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 the segments 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 SEARCH procedure that the algorithm uses to compute the claimer of v outputs 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 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 Fig. 41(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 Fig. 43(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 $d(claimer(u), u)$, 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. □

The above lemma completes the proof of the correctness of our algorithm. Now we show that the total number of the processed candidate events is only linear. 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; denote by t_j, for $1 \le j \le m$, the time at which W reaches v_j. Note that if the simulation is restarted because of a vertex event at v_j, then, by Lemma 5.9, the simulation is restarted exactly from time t_j—that is, t_j depends only on W and on the previous true candidate events. Note also that the simulation is only restarted from times $t_1, \ldots, t_m$.

Lemma 5.10 *When the vertex events at vertices $v_1, \ldots, v_k$, for $1 \le k \le m$, 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 $t' \ge t$, 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 that 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 $v_1, \dots, v_{k-1}$, and consider the version W_k of W that is created at time t_k when the vertex events at vertices $v_1, \dots, v_k$ 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 $\{v_1, \dots, v_{k-1}\}$. But then, restarting the propagation from time t_v contradicts the induction hypothesis. □

Lemma 5.11 *For each $1 \le j \le m$, the simulation is restarted from time t_j at most 2^{j-1} times.*

Proof By induction on j. By Lemma 5.10, the simulation is restarted from time t_1 at most once. Now assume that $j \ge 2$ and that the claim is true for times $t_1, \dots, t_{j-1}$.

By Lemma 5.9, the vertex event at v_j is eventually processed at time t_j; by Lemma 5.10, there are no further restarts from time t_j after we get a version of W that encodes all the events at $v_1, \dots, v_j$. 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 $t_1, \dots, t_{j-1}$ 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. □

Remark From a practical point of view, the algorithm can be significantly 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.

By Lemma 5.11, 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 x has at most $O(1)$ "identical copies," which are the same event, processed at the same location (and at the same simulation time t_x) after different simulation restarts. At most one of these copies of x remains encoded in valid versions of W, and the rest are discarded (that is, there is at most one valid version of W that has been created at simulation time t_x to reflect the correct processing of x, and the following valid versions of W are coherent with this version). 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 distinct 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 $2|f|$.

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

Proof First we bound the total number of waves that are created 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 $R(e)$), 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 5.11 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 5.11, 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.

Consider a candidate event of type (iv) at a location x at time t_x, in some fixed $T_B(e)$, and let f denote the transparent edge of $\mathcal{C}$ that is crossed by the wave from the generator s' eliminated at x. Denote by d_1 (resp., d_2) the distance along $\pi(s', x)$ from

s' to f (resp., from f to x); that is, $d_2 > 2|f|$ and $d_1 + d_2 = d(s', x) = t_x$. Before the update of $t_{\text{stop}}(f)$, caused by the processing of this event, the value of $t_{\text{stop}}(f)$ must have been equal to 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 not have been 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 5.11 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 $k = O(1)$ cells away from the cell c. Therefore the event at x is detected during the simulation in the cell which contains x, where the event at x is considered a *true event*, and during the simulation in at most k other cells; hence, by Lemma 5.12, the total number of these candidate events during the whole algorithm is bounded by $O(kn) = O(n)$. □

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

Preprocessing Building Blocks Let B be a building block of a surface cell c. A generator of a wavefront W is called *active in B* if it was detected by the algorithm to be involved in a bisector event inside B. 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 B that was covered by 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.[14] 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 Fig. 46 for an illustration.

[14] 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 of 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 Sect. 3.1), the split portions cannot collide with each other inside B; see Fig. 46.

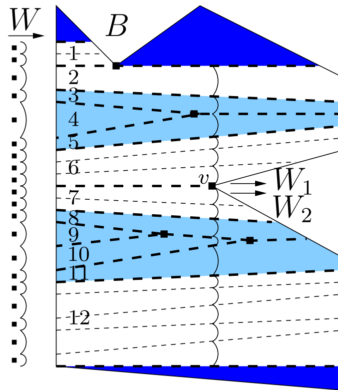

Fig. 46 The wavefront W enters the building block B (in this example, B is a nonconvex block of type I, bounded by *solid lines*) from the left. The partition $local(W, B)$ is drawn by thick *dashed lines*; *thin dashed lines* denote bisectors of W that lie fully in the interior of the inactive regions. 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 $local(W, B)$. 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, within B, 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).

Note that the complexity of $local(W, B)$ is $O(k + 1)$, where k is the number of true critical (bisector and vertex) events of W in B. The partition can actually be computed "on the fly" during the propagation of W in B, in additional $O(k)$ time.

We preprocess each such partition $local(W, B)$ for point location [13, 22], 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 the portion of W corresponding to the inactive region, in $O(\log n)$ time (see Sect. 5.1).

Preprocessing S_{3D} In order to locate the cell of S 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 [10]. 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 Sect. 2.4) to find the facet f of ∂P that contains p in its closure.
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 Sect. 2.4).
   (b) Calculate the portion $c \cap f$ and determine whether p lies in that portion.

   If p is contained in more than one surface cell, assign it to an arbitrary cell among them.
4. Among the $O(1)$ building blocks of c, find a block B that contains p. For each wavefront W that has traversed 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 such claimers of p.
5. 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. In case there are several such paths, each can be reported in the same manner.

Steps 1–3 take $O(\log n)$ time, using [10] and the data structure defined in Sect. 2.4. As argued above, it takes $O(\log n)$ time to perform Step 4. This 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 briefly sketches the proof of Theorem 2.1, by describing an algorithm for constructing a conforming 3D-subdivision for a set V of n points in $\mathbb{R}^3$. Since this is a straightforward generalization of the construction of a similar conforming subdivision in the plane [18], we only describe the details that are different from those in [18], and provide a few necessary definitions.

The main part of the algorithm constructs a *1-conforming* 3D-subdivision S^1_{3D} of size $O(n)$ in $O(n\log n)$ time, which is then transformed into a conforming 3D-subdivision S_{3D} by subdividing each face of S^1_{3D} into 16×16 square subfaces, in $O(n)$ additional time.

Constructing the 1-Conforming 3D-Subdivision 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 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 $\mathcal{G}_{i+2}$. 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; see Fig. 47. 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 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 i, only a minimal set $\mathcal{Q}(i)$ of quads is maintained. 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). 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*.

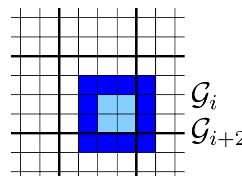

Fig. 47 The planar analog of an i-quad (*darkly shaded*) and its core (*lightly shaded*)

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*.

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. These tasks are performed by the procedures *Growth* and *Build-subdivision*, respectively. We omit the description of *Growth* (which is a duplicate three-dimensional version of the same procedure in [18]), but briefly review some of its features, to facilitate the description of *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 the core of a member of *Growth*(S). Let *Growth*(q), or $\tilde{q}$, denote the unique $(i+2)$-quad returned by the procedure *Growth* with input q (see [18, 34] for details concerning the choice of $\tilde{q}$ among eight possible $(i+2)$-quads).

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. (This step is needed for the (MVC) property, and does not exist in [18].) 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 forming distinct simple components, even though some pairs might share boundary points.

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 $G(i)$.

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. [23] for computing an

L_∞-minimum spanning tree in three dimensions. (In the planar case of [18], the classical algorithm of Kruskal is used instead.) Then, for each edge e constructed by the algorithm, we compute the stage $k = 2\lceil \frac{1}{2}\log_2 \frac{1}{6}|e|\rceil$, 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($i-2$).

Only stages at which something happens are processed: MSF(i) changes, or there are complex components of $\mathcal{Q}(i)$ whose *Growth* computation is nontrivial. *Growth*(S) is only computed 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 at stage i.

The equivalence classes of $\mathcal{Q}(i)$ are computed by finding $k = 7^3 - 1$ nearest neighbors of each i-quad q, using the well-separated pair decomposition of [7], and by testing which of them overlaps q.[15] This is different from the planar case of [18], where the nearest-neighbors algorithm is not needed (instead, the plane is simply swept).

To recap, at each “interesting” stage i, we construct $\mathcal{Q}(i)$ from $\mathcal{Q}(i-2)$, by invoking the *Growth* procedure on the set of complex components and simple components that are about to merge with other components. As argued in [18], repeated applications of *Growth* decrease the size of $\mathcal{Q}(i)$ (specifically, after each pair of consecutive steps of *Growth*, $|\mathcal{Q}(i)|$ is at most $\frac{3}{4}$ of its previous size), until we reach a single quad containing all of V.

The running time of the L_∞-minimum spanning tree algorithm in [23] is $O(n\log n)$. The k-nearest-neighbors algorithm of [7] requires $O(m_i \log m_i + km_i) = O(m_i \log m_i)$ time to process $m_i = |\mathcal{Q}(i)|$ quads, when computing the equivalence classes of $\mathcal{Q}(i)$. As argued in [18], $\sum_i m_i = O(n)$; hence, it takes $O(n\log n)$ total time to perform this step. The space requirements of the MST construction in [23], and of the k-nearest-neighbors computation, are both $O(n)$, as well as the space requirements of the other stages of the algorithm. Other steps of the algorithm *Build-subdivision* are similar to those in [18], and therefore, 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. As shown in [18], the total cost of all the calls to *Growth* is $O(n\log n)$, and this procedure requires only linear space; hence, S_{3D} can be constructed in overall $O(n\log n)$ time, using $O(n)$ 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 with n facets in three dimensions. The algorithm takes $O(n\log n)$ preprocessing time and $O(n\log n)$ storage, and answers a shortest path query (which identifies the path and computes its length) in $O(\log n)$ time. We have used and adapted the ideas of Hershberger and Suri [18], solving Open Problem 2 of their paper, to construct “on the fly" a dynamic version of the incidence data structure of Mount [28], answering in the affirmative the question that was left open in [28].

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

As in the planar case (see [18]), our algorithm can also easily be extended to a more general instance of the shortest path problem that involves *multiple sources* on the surface of P, which is equivalent to computing their (implicit) *geodesic Voronoi diagram*. This is a partition of ∂P 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 return the shortest path length and starting direction (and, if needed, the shortest path itself) from s to q; this 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.

It is natural to extend the wavefront propagation method to the shortest path problem on the surface of a *nonconvex* polyhedral surface. As our more recent results [33] show, such an extension, which still runs in optimal $O(n\log n)$ time, exists for several restricted classes of "realistic" polyhedra, such as a polyhedral terrain whose maximal facet slope is bounded, and a few other classes. However, the question of whether a subquadratic-time algorithm exists for the most general case of nonconvex polyhedra, remains open.

Finally, we conclude with two less prominent open problems.

1. Can the space complexity of the algorithm be reduced to linear? Note that our $O(n\log n)$ storage bound is a consequence of only the need to perform path copying to ensure persistence of the surface unfolding data structure in Sect. 2.4 and the source unfolding data structure in Sect. 5.1. Note also that the related algorithms of [18] and [28] also use $O(n\log n)$ storage.
2. Can an unfolding of a surface cell of S overlap itself?

Acknowledgements 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 invaluable help in designing the data structures, to Joe Mitchell for his comments of Kapoor's paper, and to an anonymous referee for a careful review and many useful suggestions. We also acknowledge, with thanks, a recent correspondence with Sanjiv Kapoor concerning some of the details in his paper.

References

1. Agarwal, P.K., Aronov, B., O'Rourke, J., Schevon, C.A.: Star unfolding of a polytope with applications. SIAM J. Comput. **26**, 1689–1713 (1997)
2. Agarwal, P.K., Har-Peled, S., Sharir, M., Varadarajan, K.R.: Approximate shortest paths on a convex polytope in three dimensions. J. ACM **44**, 567–584 (1997)
3. Aleksandrov, L., Lanthier, M., Maheshwari, A., Sack, J.-R.: An ϵ-approximation algorithm for weighted shortest paths on polyhedral surfaces. In: 6th Scand. Workshop Algorithm Theory. Lecture Notes in Computer Science, vol. 1432, pp. 11–22. Springer, Berlin (1998)
4. Aleksandrov, L., Maheshwari, A., Sack, J.-R.: An improved approximation algorithm for computing geometric shortest paths. In: 14th FCT. Lecture Notes in Computer Science, vol. 2751, pp. 246–257. Springer, Berlin (2003)
5. Aloupis, G., Demaine, E.D., Langerman, S., Morin, P., O'Rourke, J., Streinu, I., Toussaint, G.: Unfolding polyhedral bands. In: Proc. 16th Canad. Conf. Comput. Geom., pp. 60–63 (2004)

6. Aronov, B., O'Rourke, J.: Nonoverlap of the star unfolding. Discrete Comput. Geom. **8**, 219–250 (1992)
7. Callahan, P.B., Kosaraju, S.R.: A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. J. ACM **42**(1), 67–90 (1995)
8. Chen, J., Han, Y.: Shortest paths on a polyhedron, part I: computing shortest paths. Int. J. Comput. Geom. Appl. **6**, 127–144 (1996)
9. Chen, J., Han, Y.: Shortest paths on a polyhedron, part II: storing shortest paths. Tech. Rept. 161-90, Comput. Sci. Dept., Univ. Kentucky, Lexington, KY, February 1990
10. de Berg, M., van Kreveld, M., Snoeyink, J.: Two- and three-dimensional point location in rectangular subdivisions. J. Algorithms **18**, 256–277 (1995)
11. Demaine, E.D., O'Rourke, J.: Geometric Folding Algorithms: Linkages, Origami, and Polyhedra. Cambridge University Press, Cambridge (2007)
12. Driscoll, J.R., Sleator, D.D., Tarjan, R.E.: Fully persistent lists with catenation. J. ACM **41**(5), 943–949 (1994)
13. Edelsbrunner, H., Guibas, L.J., Stolfi, J.: Optimal point location in a monotone subdivision. SIAM J. Comput. **15**, 317–340 (1986)
14. Guibas, L., Hershberger, J., Leven, D., Sharir, M., Tarjan, R.E.: Linear time algorithms for visibility and shortest path problems inside simple polygons. Algorithmica **2**, 209–233 (1987)
15. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Proc. 19th IEEE Sympos. Found. Comput. Sci., pp. 8–21 (1978)
16. Har-Peled, S.: Approximate shortest paths and geodesic diameters on convex polytopes in three dimensions. Discrete Comput. Geom. **21**, 216–231 (1999)
17. Har-Peled, S.: Constructing approximate shortest path maps in three dimensions. SIAM J. Comput. **28**(4), 1182–1197 (1999)
18. Hershberger, J., Suri, S.: 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., pp. 508–517 (1993); Manuscript, Washington Univ., St. Louis (1995)
19. Hershberger, J., Suri, S.: Practical methods for approximating shortest paths on a convex polytope in $\mathbb{R}^3$. Comput. Geom. Theory Appl. **10**(1), 31–46 (1998)
20. Italiano, G.F., Raman, R.: Topics in Data Structures. In: Atallah, M.J. (ed.) Handbook on Algorithms and Theory of Computation, Chap. 5. CRC Press, Boca Raton (1998)
21. Kapoor, S.: Efficient computation of geodesic shortest paths. In: Proc. 32nd Annu. ACM Sympos. Theory Comput., pp. 770–779 (1999)
22. Kirkpatrick, D.: Optimal search in planar subdivisions. SIAM J. Comput. **12**, 28–35 (1983)
23. Krznaric, D., Levcopoulos, C., Nilsson, B.J.: Minimum spanning trees in d dimensions. Nord. J. Comput. **6**(4), 446–461 (1999)
24. Lanthier, M., Maheshwari, A., Sack, J.-R.: Approximating shortest paths on weighted polyhedral surfaces. Algorithmica **30**(4), 527–562 (2001)
25. Mata, C., Mitchell, J.S.B.: A new algorithm for computing shortest paths in weighted planar subdivisions. In: Proc. 13th Annu. ACM Sympos. Comput. Geom., pp. 264–273 (1997)
26. Mitchell, J.S.B., Mount, D.M., Papadimitriou, C.H.: The discrete geodesic problem. SIAM J. Comput. **16**, 647–668 (1987)
27. Mount, D.M.: On finding shortest paths on convex polyhedra. Tech. Rept., Computer Science Dept., Univ. Maryland, College Park, October 1984
28. Mount, D.M.: Storing the subdivision of a polyhedral surface. Discrete Comput. Geom. **2**, 153–174 (1987)
29. O'Rourke, J.: Computational geometry column 35. Int. J. Comput. Geom. Appl. **9**, 513–515 (1999); also in SIGACT News, 30(2), 31–32 (1999)
30. O'Rourke, J.: On the development of the intersection of a plane with a polytope. Tech. Rept. 068, Smith College, June 2000
31. O'Rourke, J., Suri, S., Booth, H.: Shortest paths on polyhedral surfaces. Manuscript, The Johns Hopkins Univ., Baltimore, MD (1984)
32. Paul, R.P.: Robot Manipulators: Mathematics, Programming, and Control. MIT Press, Cambridge (1981)
33. Schreiber, Y.: Shortest paths on realistic polyhedra. In: Proc. 23rd Annu. ACM Sympos. Comput. Geom., pp. 74–83 (2007)
34. Schreiber, Y., Sharir, M.: An optimal-time algorithm for shortest paths on a convex polytope in three dimensions, http://www.tau.ac.il/~michas/ShortestPath.pdf. Also in Y. Schreiber, PhD thesis, http://www.tau.ac.il/~syevgeny/SchreiberThesis.pdf

35. Sharir, M.: On shortest paths amidst convex polyhedra. SIAM J. Comput. **16**, 561–572 (1987)
36. Sharir, M., Schorr, A.: On shortest paths in polyhedral spaces. SIAM J. Comput. **15**, 193–215 (1986)
37. Tarjan, R.E.: Data structures and network algorithms. SIAM CBMS, p. 44 (1983)
38. Varadarajan, K.R., Agarwal, P.K.: Approximating shortest paths on a nonconvex polyhedron. In: Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci., pp. 182–191 (1997)
39. Weisstein, E.W.: Riemann surface. MathWorld—a Wolfram web resource. http://mathworld.wolfram.com/RiemannSurface.html