Computing the Volume of the Union of Cubes*

Pankaj K. Agarwal
Department of Computer Science
Duke University

Haim Kaplan
School of Computer Science
Tel Aviv University

Micha Sharir
School of Computer Science
Tel Aviv University
and
Courant Institute
New York University

ABSTRACT

Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$, and let $\mathcal{U}(\mathcal{C})$ denote the union of $\mathcal{C}$. We present an algorithm that can compute the volume of $\mathcal{U}(\mathcal{C})$ in time $O(n^{4/3}\log n)$. The previously best known algorithm, by Overmars and Yap, computes the volume of the union of any n axis-aligned boxes in $\mathbb{R}^3$ in $O(n^{3/2}\log n)$ time.

Categories and Subject Descriptors

F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations

General Terms

Algorithms, Theory

Keywords

Union of objects, Klee's measure problem, arrangements, segment trees.

1. INTRODUCTION

Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$, and let $\mathcal{U}(\mathcal{C})$ denote the union of $\mathcal{C}$. The problem studied in this paper is to compute the volume of $\mathcal{U}(\mathcal{C})$ efficiently. This is related to the well-known *Klee's measure problem*. In 1977 Victor Klee [11] had presented an $O(n\log n)$ time algorithm for computing the union of n intervals in $\mathbb{R}^1$ and had asked whether his algorithm was optimal. An $\Omega(n\log n)$ lower bound was proved by Fredmen and Weide [9]. Bentley [4] studied the two-dimensional version of Klee's measure problem. When extended to computing the volume of the union of n d-dimensional axis-aligned boxes, the running time of Bentley's algorithm is $O(n^{d-1}\log n)$. Later, van Leeuwen and Wood [10] improved the running time to $O(n^2)$ for $d=3$. The problem lay dormant for a while until Overmars and Yap presented an algorithm with $O(n^{d/2}\log n)$ running time [12]. Despite several attempts, no further progress was made on this problem, except for a somewhat simpler solution but with the same running time for $d=3,4$ [7], and for a more space-efficient algorithm [6]. See [1] for a brief history of the problem.

In contrast, there has been significant progress in the last decade on obtaining sharp bounds on the combinatorial complexity of the union (i.e., the number of faces of all dimensions on the boundary of the union) of axis-aligned (and other) objects. For example, Boissonnat *et al.* [5] proved that the combinatorial complexity of n axis-aligned cubes in $\mathbb{R}^d$ is $\Theta(n^{\lceil d/2\rceil})$, and it is $\Theta(n^{\lfloor d/2\rfloor})$ if all the cubes have the same size. Note that this bound is considerably better than the $\Theta(n^d)$ worst-case bound on the union of n axis-aligned boxes in $\mathbb{R}^d$. This suggests that it might be easier to compute the volume of the union of n cubes. Indeed, the volume of the union of n unit cubes in $\mathbb{R}^3$ can be computed in $O(n\log n)$ time, by computing their union explicitly (which has linear complexity). However this will not lead to an efficient algorithm for cubes of different sizes. Edelsbrunner [8] gave an inclusion-exclusion formula for computing the volume of the union of n balls (see also [2]). It might be possible to extend his approach to computing the volume of the union of cubes in $\mathbb{R}^3$, but the running time will be $\Omega(n^2)$ in the worst case.

In this paper we show that one can indeed exploit the special structure of cubes in $\mathbb{R}^3$ in order to compute the volume of their union more efficiently, and present the following result.

THEOREM 1.1. *Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$. The volume of $\mathcal{U}(\mathcal{C})$ can be computed in time $O(n^{4/3}\log n)$.*

The high-level approach of our algorithm is similar to that of [12], in the sense that it is also based on sweeping the space with a horizontal plane. The details are, however, more intricate, and exploit, sometimes in subtle ways, the fact that we are dealing with cubes, rather than boxes. We believe that our algorithm is far from being optimal, and that the running time can be improved to $O(n\,\mathrm{polylog}(n))$, but so far we have not been able to overcome all of the technical difficulties (which, as the reader might appreciate, are quite numerous).

The algorithm asserted in Theorem 1.1 is presented in the three following sections. Section 2 gives the high-level description of the algorithm, Section 3 goes into the more technical low-level de-

*Work by Pankaj Agarwal and Micha Sharir was supported by a grant from the U.S.-Israel Binational Science Foundation. Work by Pankaj Agarwal was also supported by by NSF under grants CCR-00-86013, EIA-01-31905, and CCR-02-04118, and an ARO grant W911NF-04-1-0278. Work by Haim Kaplan was supported by Grant 975/06 from the Israel Science Fund. Work by Micha Sharir was also supported by NSF Grants CCR-00-98246 and CCF-05-14079, by Grant 155/05 from the Israel Science Fund, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

SCG'07, June 6–8, 2007, Gyeongju, South Korea.

tails, and Section 4 presents further details of the data structure that maintains the union during the sweep.

2. THE GLOBAL STRUCTURE

We assume that the given cubes are in *general position*. In particular, we assume that no plane support facets of two distinct cubes in $\mathcal{C}$. Let $z_1 < \cdots < z_{2n}$ be the (distinct) z-coordinates of the vertices of cubes in $\mathcal{C}$, sorted in increasing order. We sweep a horizontal plane Π in the $(+z)$-direction from $-\infty$ to $+\infty$, stopping at each z_i. Let $\Pi(t)$ denote the horizontal plane at $z = t$. For each $1 \le i < 2n$, the cross-section $\mathcal{U}(\mathcal{C}) \cap \Pi(z)$ is the same for all $z \in (z_i, z_{i+1})$. Let a_i denote the area of this cross-section. Then

$$\operatorname{Vol}\mathcal{U}(\mathcal{C}) = \sum_{i=1}^{2n-1} a_i(z_{i+1} - z_i).$$

We thus need to maintain a_i as we sweep the horizontal plane. The intersection of $\Pi(z)$ with $\mathcal{U}(\mathcal{C})$ is the union of a set $\mathcal{S}$ of squares that changes dynamically—a square is added to the intersection when Π sweeps through the bottom facet of its corresponding cube, and is removed from the intersection when Π sweeps through the top facet of its cube. We describe a data structure that maintains, in $O(n^{1/3}\log n)$ amortized time, the area of the union of $\mathcal{S}$, denoted by $\operatorname{Area}\mathcal{U}(\mathcal{S})$, as we insert a square into $\mathcal{S}$ or delete a square from $\mathcal{S}$ during the sweep. This implies Theorem 1.1. Our procedure exploits, in a crucial though subtle way, the special (obvious) property that the *life-time* of a square of side length h (regarding the z-direction as "time") is also h time units.

In our case, we know in advance the set $\mathcal{V} \subset \mathbb{R}^2$ of vertices of all the squares that will ever be inserted into $\mathcal{S}$. That is, $\mathcal{V}$ is the set of the xy-projections of the vertices of the given cubes, and we have $|\mathcal{V}| = 4n$. Let $\mathcal{B}$ be the smallest axis-parallel rectangle containing $\mathcal{V}$. We choose the parameter $s = n^{1/3}$, and partition $\mathcal{B}$ into s rectangles $B_1, \ldots, B_s$, called *slabs*, by vertical lines (parallel to the y-axis), so that the interior of each B_i contains at most $4n/s = O(n^{2/3})$ vertices of $\mathcal{V}$; see Figure 1. Next, we partition each B_i into s rectangles, called *cells*, by lines parallel to the x-axis, so that the interior of each cell contains at most $4n/s^2 = O(n^{1/3})$ vertices. For $1 \le i \le s$, we maintain $\alpha_i = \operatorname{Area}\mathcal{U}(\mathcal{S}) \cap B_i$, using a binary tree $\mathcal{T}_i$ with s leaves. Each node v of $\mathcal{T}_i$ is associated with a rectangle $\square_v$ contained in B_i and touching its two vertical sides. For the ith leftmost leaf v of $\mathcal{T}_i$, $\square_v$ is the ith bottom-most cell of B_i. For an interior node v with children w and z, $\square_v = \square_w \cup \square_z$. For a node $v \in \mathcal{T}_i$, let $\mathcal{S}_v \subseteq \mathcal{S}$ be the set of squares whose boundaries intersect the interior of $\square_v$, and $\mathcal{S}_v^* \subseteq \mathcal{S}$ be the set of squares that contain $\square_v$ but not $\square_{p(v)}$ (where $p(v)$ is the parent of v). At each leaf v of $\mathcal{T}_i$ we store the respective set $\mathcal{S}_v$, and we also store $\sigma_v = |\mathcal{S}_v^*|$ at each node v. For a node $v \in \mathcal{T}_i$, let $\alpha_v = \operatorname{Area}\mathcal{U}(\mathcal{S}_v \cup \mathcal{S}_v^*) \cap \square_v$. If v is a leaf, we compute and update α_v using the algorithm described in Section 3. For an interior node v with children w and z, we have

$$\alpha_v = \begin{cases} \operatorname{Area}\square_v & \text{if } \sigma_v \ge 1, \\ \alpha_w + \alpha_z & \text{if } \sigma_v = 0. \end{cases} \tag{1}$$

When we insert a square S, we first find all the slabs B_i that S meets. Then, for each of these B_i, we find the leaves v of $\mathcal{T}_i$ such that $\square_v \cap \partial S \neq \emptyset$. For each such v, we insert S into $\mathcal{S}_v$ and update α_v using the algorithm described in Section 3. If B_i does not contain a vertex of S, then S is inserted into at most two leaves w and z, such that $\square_w$ and $\square_z$ intersect the horizontal edges of S. Next, we find all $O(\log n)$ nodes u in $\mathcal{T}_i$ which lie between w and z, and whose parents lie along the two paths of $\mathcal{T}_i$ to w and to z, so that $\square_u \subseteq S$ but $\square_{p(u)} \not\subseteq S$. For each such u, we increment the value of σ_u and set $\alpha_u = \operatorname{Area}\square_u$. If B_i contains a vertex of S, then S may have to be inserted into many (perhaps all) leaves of $\mathcal{T}_i$, and $S \notin \mathcal{S}_v^*$ for any $v \in \mathcal{T}_i$. For each leaf v for which $\square_v \cap \partial S \neq \emptyset$, we update α_v, using the algorithm of Section 3.

Finally, using (1) in a bottom-up manner, we update the values α_u for all ancestors u of any node v that has been updated. We repeat this procedure for each of the slabs B_i, and return the value of $\sum_{i=1}^{s} \alpha_{\operatorname{root}(\mathcal{T}_i)}$. A square is deleted from $\mathcal{S}$ in a similar manner.

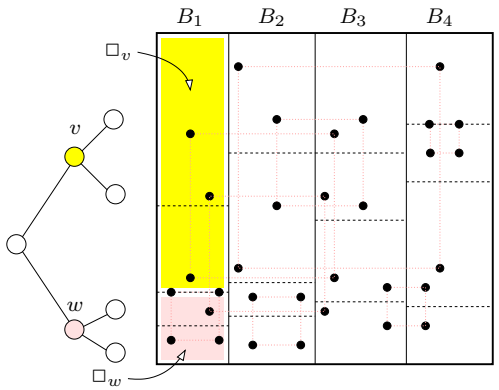

Figure 1. Partition of $\mathcal{B}$ into four slabs and sixteen cells, and the tree $\mathcal{T}_1$.

Let v be a leaf of $\mathcal{T}$ such that $S \in \mathcal{S}_v$. We show in the next section (cf. Lemma 3.1) that if $\square_v \cap \partial S \neq \emptyset$, then

(i) α_v can be updated in $O(\log n)$ amortized time provided that $\square_v$ does not contain a vertex of S in its interior, and

(ii) α_v can be updated in $O((n/s^2)\log n) = O(n^{1/3}\log n)$ amortized time if $\square_v$ does contain a vertex of S in its interior.

There are at most four cells that contain a vertex of S, and there are at most $4s$ cells that intersect ∂S, so we spend a total of $O((s + n/s^2)\log n) = O(n^{1/3}\log n)$ amortized time in updating the areas at the leaves of the trees $\mathcal{T}_i$. We then update the ancestors of the updated leaves. In the worst case, we visit all the nodes of at most two $\mathcal{T}_i$'s—if B_i contains a vertex of S—and $O(\log n)$ nodes of any other $\mathcal{T}_i$. We spend $O(1)$ time at each of these ancestor nodes. Hence, the total amortized time spent in updating $\operatorname{Area}\mathcal{U}(\mathcal{S})$ when a square is inserted or deleted is $O(n^{1/3}\log n)$. That is, $\operatorname{Area}\mathcal{U}(\mathcal{S})$ can be updated in $O(n^{1/3}\log n)$ amortized time after each update operation, and Theorem 1.1 follows.

Remark. We believe that the running time of the algorithm can be improved to $O(n\,\mathrm{polylog}(n))$ by using a recursive binary partioning of $\mathcal{B}$ and maintaining a similar (but more involved) information at each cell in the recursive partition. However we have not succeeded in overcoming all the technical difficulties in updating the information at each cell in amortized $O(\mathrm{polylog}(n))$ time when a square is inserted or deleted.

3. MANTAINING THE UNION WITHIN A CELL

Let $\square = [x_0, x_1] \times [y_0, y_1]$ be a fixed cell in the partition of $\mathcal{B}$, which, as we recall, is an axis-parallel rectangle in $\mathbb{R}^2$. Without loss of generality, we assume that $x_1 - x_0 \ge y_1 - y_0$ (handling cells with $x_1 - x_0 < y_1 - y_0$ is done in a symmetric manner, switching the roles of the x- and y-axes). Let $p_0 = (x_0, y_0), p_1 = (x_1, y_0), p_2 =$

(x_1, y_1), and $p_3 = (x_0, y_1)$ be its vertices in counterclockwise order. Let $\mathcal{S}$ be a set of squares whose boundaries intersect $\square$. We describe a data structure for maintaining the area $\alpha_\square$ of $\mathcal{U}(\mathcal{S}) \cap \square$ under insertions and deletions of squares to/from $\mathcal{S}$, where we also assume that the life-span of each square in $\mathcal{S}$ is equal to its side length. Recall that the set $\mathcal{S}$ is updated when the sweep plane passes through a top or a bottom facet of a cube in $\mathcal{C}$.

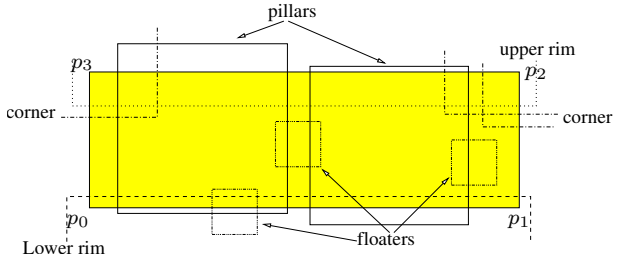

Figure 2. Partition of $\mathcal{S}$ into various categories.

Since the boundary of each square in $\mathcal{S}$ intersects $\square$, and the x-span of $\square$ is at least as large as its y-span, no square $S \in \mathcal{S}$ can intersect both left and right edges of $\square$ without fully containing either the top or bottom edge of $\square$. We partition $\mathcal{S}$ into the following subsets (we assume that no square of $\mathcal{S}$ fully contains $\square$—this situation is handled in the high-level description given above); see Figure 2):

Upper rim. The set of squares, denoted by $\mathbb{U}$, that contain the top edge of $\square$. We store $\mathbb{U}$ in a list sorted in decreasing order of the y-coordinates of their bottom edges.

Lower rim. The set of squares, denoted by $\mathbb{L}$, that contain the bottom edge of $\square$. We store $\mathbb{L}$ in a list sorted in increasing order of the y-coordinates of their top edges.

Pillars. The set of squares, denoted by $\mathbb{P}$, that intersect both top and bottom edges of $\square$.

Corners. The set of squares, denoted by $\mathbb{C}$, that contain exactly one vertex of $\square$; exactly one vertex of each corner square lies in $\square$.

Floaters. The set of remaining squares, denoted by $\mathbb{F}$; at least two (i.e., either two or four) of the vertices of each floater square lie in $\square$.

The first three types of squares are called *long*, and the last two types are called *short*.[1] The *floor* of $\square$ is the top edge of the last square in the lower rim (i.e., the highest edge in the lower rim), and the *ceiling* is the bottom edge of the last square (the lowest edge) in the upper rim.

Returning to our overall algorithm described in the previous section, $\square$ has only $O(n^{1/3})$ short squares, so we can spend time linear in the number of short squares to indert or delete a short square. For example, we can afford to (and indeed we will) recompute the union of the corners or of the floaters when we insert or delete one of these squares. In contrast, inserting/deleting a long square (rim or pillar) is more challenging, since we want to do it in only $O(\log n)$ (amortized) time.

The main technical complication in our solution is that, while it is fairly easy to maintain the area of the union of each class of squares separately, it is much more involved to maintain the area of the combined union. Our approach is to maintain this latter area as the sum of areas of disjoint portions of $\square$—the area covered by the rims, the area covered by the pillars but not by the rims, the area covered by the corner squares but not by the pillars or rims, and finally the area covered by the floaters but not by any other square. Maintaining these disjoint areas is somewhat tricky; the high-level details are given in this section, and the low-level details in Section 4.

We call a (rectilinear) polygon *staircase* if it consists of a rectilinear chain that is both x- and y-monotone (in each coordinate it can be either increasing or decreasing), and the endpoints of the chain are connected together by a horizontal and a vertical edge; see Figure 3 (i). The common endpoint of the horizontal and the vertical edge is called the *apex* of the polygon. Since the complexity of the union of a set of axis-parallel squares is linear, $\mathcal{U}(\mathbb{C}) \cap \square$ has $O(|\mathbb{C}|)$ vertices. We can decompose $\mathcal{U}(\mathbb{C}) \cap \square$ into four pairwise-disjoint staircase polygons, P_0, P_1, P_2, P_3, with a total of $O(|\mathbb{C}|)$ vertices, such that the apex of P_i is the vertex p_i of $\square$ (see Figure 3 (ii)). Informally, P_i is composed of the corner squares that contain p_i, but since other squares can "nibble off" some portions of these squares, as is illustrated in the figure, P_i may be smaller than the union of its squares. Also, the P_i's are not uniquely defined, but, since they will be recomputed from scratch when we insert or delete a square into $\mathbb{C}$, the precise way of defining them does not matter, as long as we keep them disjoint. We decompose each P_i into a set $\tilde{\mathbb{C}}_i$ of rectangles by computing the vertical decomposition of P_i, i.e., drawing a vertical edge from each reflex vertex of P_i within $\square$ until it touches a horizontal edge of $\square$; see Figure 3 (iii). Set $\tilde{\mathbb{C}} = \bigcup_{i=0}^{3} \tilde{\mathbb{C}}_i$. By construction, $\mathcal{U}(\tilde{\mathbb{C}}) = \mathcal{U}(\mathbb{C}) \cap \square$.

We compute $(\mathcal{U}(\mathbb{F}) \cap \square) \setminus \mathcal{U}(\mathbb{C})$, the portion of $\mathcal{U}(\mathbb{F}) \cap \square$ that lies outside $\mathcal{U}(\mathbb{C})$, and partition it into pairwise-disjoint rectangles by computing its vertical decomposition. Let $\mathcal{R} = \{R_1, \ldots, R_u\}$ be the set of the rectangles in the resulting decomposition. Since $\mathcal{U}(\mathbb{F})$ and $\mathcal{U}(\mathbb{C})$ have $O(|\mathbb{F}|)$ and $O(|\mathbb{C}|)$ vertices, respectively, $|\mathcal{R}| = O(|\mathbb{C}| + |\mathbb{F}|)$. See Figure 4. We call a rectangle of $\mathcal{R}$ *stalactite* (resp., *stalagmite*) if it intersects the ceiling (resp., floor) of $\square$; a rectangle may be both a stalactite and a stalagmite. Let $\mathbb{Sg}$ (resp., $\mathbb{Sc}$) denote the set of stalagmites (resp., stalactites) in $\mathcal{R}$. We store the horizontal edges of $\mathbb{F}$ (or, more precisely, the rectangles in $\mathcal{R}$) in a list Λ, sorted by their y-coordinates.

In addition we maintain the following information.

fl: The y-coordinate of the floor of $\square$.

cl: The y-coordinate of the ceiling of $\square$.

π: The length of the portion of the top edge of $\square$ (or any other horizontal line intersecting $\square$) covered by the pillars.

ψ: The area of $\mathcal{U}(\tilde{\mathbb{C}})$ not covered by the long squares (i.e., pillars, upper rim, and lower rim).

φ: The area of $\mathcal{U}(\mathcal{R})$ not covered by the long squares.

Assuming that we can maintain the above data as the squares of $\mathcal{S}$ are inserted and deleted, the area $\alpha_\square$ (which, as we recall, is the area of the portion of $\square$ covered by the squares of $\mathcal{S}$, whose boundaries cross $\square$) can be computed as follows. If $\mathrm{cl} \le \mathrm{fl}$, then $\alpha_\square$ equals the entire area of $\square$. Otherwise,

$$\alpha_\square = (x_1 - x_0)[(\mathrm{fl} - y_0) + (y_1 - \mathrm{cl})] + (\mathrm{cl} - \mathrm{fl})\pi + \psi + \varphi. \quad (2)$$

Indeed, the first term is the area of the region covered by the upper rim squares and the lower rim squares. The second term is the area of the region covered by the pillars but not by any rim square. The

[1] This is somewhat of a misnomer for corner squares, which can be quite large compared with $\square$, but we still think of them as short since they have a vertex inside $\square$.

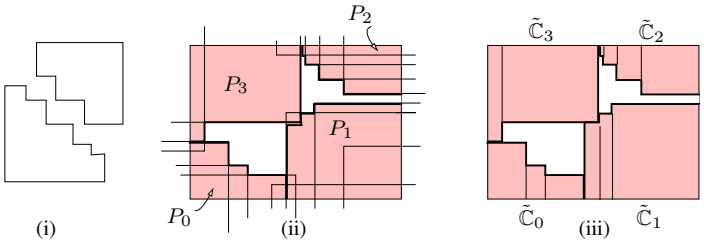

Figure 3. (i) Staircase polygons. (ii) Decomposition of $\mathcal{U}(\mathbb{C})$ into four staircase polygons $P_0, \ldots, P_3$. (iii) The decomposition $\tilde{\mathbb{C}}_i$ of each P_i into rectangles.

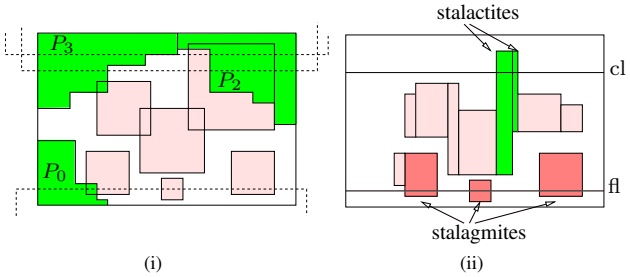

Figure 4. (i) Squares in $\mathbb{F}$; the darkly-shaded region is $\mathcal{U}(\mathbb{C}) \cap \square$, and the lightly-shaded region is $(\mathcal{U}(\mathbb{F}) \cap \square) \setminus \mathcal{U}(\mathbb{C})$; the rim squares are drawn as dashed. (ii) Rectangles in $\mathcal{R}$; the dark shaded rectangles are the stalagmites and stalactites;

third term is the area of the region covered by the corner squares but not by any rim square or pillar, and the fourth term is the area of the region covered by the floater squares but not by any other type of squares (recall that, by construction, $\mathcal{R}$ is disjoint from the corner squares). See Figure 5.

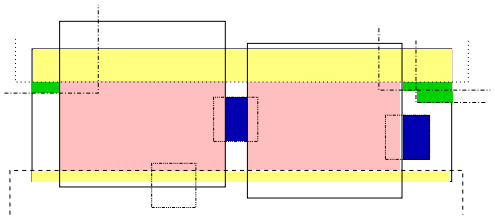

Figure 5. Each term of (2) is shown in a different shade.

We maintain π using a segment tree $\mathbb{T}(\mathbb{P})$ that stores $\mathbb{P}$, we maintain ψ using a segment tree $\mathbb{T}(\tilde{\mathbb{C}})$ that stores $\tilde{\mathbb{C}}$, and we maintain φ using a segment tree $\mathbb{T}(\mathcal{R})$ that stores $\mathcal{R}$. Each of these segment trees is defined over the endpoints of the x-projections of the squares in $\mathcal{S}$ (and thus also the rectangles in $\mathbb{P}$, $\tilde{\mathbb{C}}$, and $\mathcal{R}$). Therefore, they have the same structure, and there is a bijection between the nodes of any pair of them; the only difference among them is the auxiliary information stored at each node. In principle, we can use a single segment tree in which each node contains the combined information from the individual trees $\mathbb{T}(\mathbb{P})$, $\mathbb{T}(\tilde{\mathbb{C}})$, and $\mathbb{T}(\mathcal{R})$. We treat them as three separate segment trees, though, to simplify the presentation.

We update $\mathbb{T}(\mathbb{P})$, and thereby π, when we insert or delete a pillar. Similarly, we update $\mathbb{T}(\tilde{\mathbb{C}})$ when either $\tilde{\mathbb{C}}$ changes, or when we insert or delete a pillar (even though pillars are not stored in $\mathbb{T}(\tilde{\mathbb{C}})$). The tree $\mathbb{T}(\tilde{\mathbb{C}})$ is designed to answer corner-area queries, defined as follows. Let $\Delta \subseteq [y_0, y_1]$ be a y-interval, and let W_Δ denote the rectangle $[x_0, x_1] \times \Delta$. Define

$$\psi(\Delta) = \text{Area}\left([\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta\right).$$

A *corner-area* query is: given a y-interval $\Delta \subseteq [y_0, y_1]$, compute $\psi(\Delta)$. We describe in Section 4.2 a procedure for answering such queries using $\mathbb{T}(\tilde{\mathbb{C}})$. When fl or cl changes we update ψ by performing a query on $\mathbb{T}(\tilde{\mathbb{C}})$ with $\Delta = [\text{fl}, \text{cl}]$ and setting $\psi := \psi([\text{fl}, \text{cl}])$.

Although $\mathbb{T}(\mathcal{R})$ stores only $\mathcal{R}$, the information stored at the nodes of $\mathbb{T}(\mathcal{R})$ depends on $\mathbb{P}$ and $\mathbb{C}$ too, and also on fl and cl. Therefore we update $\mathbb{T}(\mathcal{R})$ when one of the following events occurs:

(i) We insert or delete a pillar.

(ii) The set $\mathcal{R}$ changes, as we insert or delete a corner or floater square.

(iii) A rectangle in $\mathcal{R}$ becomes or stops being a stalactite or a stalagmite (due to the motion of fl or cl).

Note that

$$\varphi = \text{Area}\left([\mathcal{U}(\mathcal{R}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta\right),$$

where $\Delta = [\text{fl}, \text{cl}]$, and W_Δ is as defined above. It is easy to compute φ in $O(1)$ time, using the values of fl and cl, and the information maintained at the root of $\mathbb{T}(\mathcal{R})$; see Section 4.3. We update φ after every update to $\mathbb{T}(\mathcal{R})$ and after inserting or deleting a lower or an upper rim square.

We show in Section 4 how to implement these segment trees so that (i) inserting a rectangle into, or deleting a rectangle from $\mathbb{T}(\mathbb{P})$, $\mathbb{T}(\tilde{\mathbb{C}})$, or $\mathbb{T}(\mathcal{R})$ takes $O(\log n)$ time; (ii) a corner-area query to $\mathbb{T}(\tilde{\mathbb{C}})$ takes $O(\log n)$ time.

Inserting/deleting a short square. Suppose we want to insert a short square S into $\mathcal{S}$ or delete S from $\mathcal{S}$. Let $\mu \leq 4n/s^2 = O(n^{1/3})$ denote the maximum number of short squares ever present in $\mathcal{S}$. If $S \in \mathbb{C}$, we recompute $\tilde{\mathbb{C}}$ in $O(\mu \log n)$ time. We delete the old rectangles of $\tilde{\mathbb{C}}$ from $\mathbb{T}(\tilde{\mathbb{C}})$ and insert the new ones, in a total of $O(\mu \log n)$ time. Next, in both cases where S is a floater or a corner square, we recompute $\mathcal{R}$ in $O(\mu \log n)$ time and reconstruct the list Λ. We delete all old rectangles of $\mathcal{R}$ from $\mathbb{T}(\mathcal{R})$ and insert each new rectangle of $\mathcal{R}$ into $\mathbb{T}(\mathcal{R})$, in a total of $O(\mu \log n)$ time. We update ψ by a corner-area query to $\mathbb{T}(\tilde{\mathbb{C}})$, and we update φ using the information at the root of $\mathbb{T}(\mathcal{R})$. Finally, using (2), we compute the new value of $\alpha_\square$. The total time spent in inserting or deleting a short square is $O(\mu \log n)$.

Inserting/deleting a long square. Suppose we want to insert a long square S. If S is a pillar, we first insert it into $\mathbb{T}(\mathbb{P})$ and then update $\mathbb{T}(\tilde{\mathbb{C}})$ and $\mathbb{T}(\mathcal{R})$. Next, we update π, ψ, and φ and

recompute $\alpha_\square$. It takes $O(\log n)$ time to update each of the trees $\mathbb{T}(\mathbb{P})$, $\mathbb{T}(\tilde{\mathbb{C}})$, and $\mathbb{T}(\mathcal{R})$, and it takes $O(\log n)$ time to perform a corner-area query on $\mathbb{T}(\tilde{\mathbb{C}})$ to obtain ψ.

If S is a lower rim square, we first insert S into the sorted sequence $\mathbb{L}$ of lower rim squares. If S is not the topmost lower rim square, it does not affect $\mathcal{U}_\square$, and we stop. Otherwise, let χ be the y-coordinate of the top edge of S. We raise the floor continuously from fl to χ, stopping at each horizontal edge of a square in $\mathbb{F}$ (or, more precisely, of a rectangle in $\mathcal{R}$) that the sweep encounters, and update $\mathbb{T}(\mathcal{R})$, until we reach χ.

In more detail, we find the first edge in Λ that lies above the current floor of $\square$, and then scan Λ from this edge upwards, processing each edge that lies below χ. Consider such an edge e at y-coordinate τ, and let $R \in \mathcal{R}$ be the rectangle bounded by e. If e is the bottom edge of R then R becomes a stalactite, so we add R to $\mathbb{S}\mathrm{c}$, we delete R from $\mathbb{T}(\mathcal{R})$ and reinsert it into $\mathbb{T}(\mathcal{R})$ as a stalactite. If R was also a stalagmite it remains a stalagmite. When the sweep reaches χ we update φ and $\alpha_\square$ in $O(1)$ time. If e is the top edge of R then R stops being a stalactite and disappears below the lower rim. So we delete R from $\mathbb{S}\mathrm{c}$ and $\mathbb{T}(\mathcal{R})$, and reinsert R into $\mathbb{T}(\mathcal{R})$ (with the new status of not being a stalactite).

If we delete the highest square of the lower rim, we essentially follow the same algorithm in reverse, lowering the floor to its new value, and processing the affected edges in Λ in the order that they get exposed. We can lower and raise the ceiling in a similar manner. If the algorithm sweeps across κ horizontal edges of rectangles in $\mathcal{R}$, then these updates take $O((\kappa+1)\log n)$ time, which is large when κ is large. Nevertheless, we show next that the *amortized* cost of these updates remains $O(\log n)$ per update.

Amortized analysis. We now show that the total time spent in lifting the floor or lowering the ceiling throughout the sweep is proportional to the overall time it takes to insert and delete all the short squares. The key observation is that the size of a lower or upper rim square is always bigger than that of any floater. Recall that a square is inserted when the sweep plane reaches the bottom facet of its cube C and is deleted when it reaches the top facet of C. Therefore if a lower or an upper rim square S is inserted *after* a floater S', then S' is deleted *before* S is deleted.

Let e be the top or bottom edge of a rectangle $R \in \mathcal{R}$. Since e is also part of an edge of a floater, it follows that the floor sweeps across e, before R is deleted, at most twice: It may be lowered below e once (during the deletion of a lower-rim square that was inserted *before* R was inserted), and then it may be raised above e once. The same is true for the ceiling. Since R has two horizontal edges, R may be reinserted into $\mathbb{T}(\mathcal{R})$ at most eight times before it disappears. (Typically, this is an overestimate: the number of times the floor or ceiling sweeps across a horizontal edge of some square in $\mathbb{S}$ is at most eight, but these edges do not have to bound the same rectangles of $\mathcal{R}$ at all these events.) Each such deletion and reinsertion of R takes $O(\log n)$ time. So if we charge R for $O(\log n)$ work when it is created, then this charge suffices to pay for all deletions and insertions of R into $\mathbb{T}(\mathcal{R})$.

A rectangle R is created when we insert or delete a short square S (either a square in $\mathbb{C}$ or a square in $\mathbb{F}$). We then compute the entire decomposition of $(\mathcal{U}(\mathbb{F}) \cap \square) \setminus \mathcal{U}(\mathbb{C})$. This decomposition has at most μ rectangles and we compute it in $O(\mu \log n)$ time. The total charges required by these new rectangles in $\mathcal{R}$ is $O(\mu \log n)$. So we can pay them by increasing the (amortized) insertion time of S by a constant factor. The following lemma follows from our discussion above and from the properties of the segment tree, which will be established in Lemma 4.1.

LEMMA 3.1. *The amortized update time of a long square in $\mathbb{S}$ is $O(\log n)$, and the worst-case update time of a short square in $\mathbb{S}$ is $O(\mu \log n)$, where $\mu \le 4n/s^2 = O(n^{1/3})$ is the maximum number of short squares in $\square$ at any time.*

Theorem 1.1 now follows from Lemma 3.1 and from the sweeping algorithm that we have described.

PI-QUERY(v, I)

if $I = \emptyset$ return 0
if $\delta_v \subseteq I$ return $\pi(v)$
if $I \subseteq \delta_v$ and $|\mathbb{P}_v| \ge 1$ return $\|I\|$
return PI-QUERY$(\mathrm{L}(v), I \cap \delta_{\mathrm{L}(v)})$
+ PI-QUERY$(\mathrm{R}(v), I \cap \delta_{\mathrm{R}(v)})$

Figure 6. Recursive procedure for computing $\pi(I)$; $\|I\|$ is the length of the interval I.

4. THE SEGMENT TREES

We now describe the procedures for maintaining $\mathbb{T}(\mathbb{P})$, $\mathbb{T}(\tilde{\mathbb{C}})$, and $\mathbb{T}(\mathcal{R})$. The leaves of each of them represent atomic intervals delimited by the x-projections of all the vertices of the squares in $\mathbb{S}$, which we know in advance, so their structure is fixed. Since these trees are isomorphic, for each node v in one of them, there are corresponding nodes in the other two trees and we will denote all of them by the same symbol v. Each node v of $\mathbb{T}(\mathbb{P})$ (and the corresponding nodes in the other two trees) is associated with an x-interval δ_v, and a rectangle $\square_v = \delta_v \times [y_0, y_1]$. If v is a leaf, then δ_v is the atomic interval associated with v. If v is not a leaf and w, z are the two children of v, then $\delta_v = \delta_w \cup \delta_z$. We denote the length of δ_v by $\|\delta_v\|$.

For an interval I, let $\mathcal{C}(I)$ denote the set of nodes v of these trees such that $\delta_v \subseteq I \subset \delta_{p(v)}$, where $p(v)$ is the parent of v, and let $\mathcal{N}(I)$ denote the set of ancestors of the nodes in $\mathcal{C}(I)$. For any interval I, $\mathcal{C}(I)$ and $\mathcal{N}(I)$ can be computed in $O(\log n)$ time. For a rectangle R, let I_R denote its x-projection. For an interior node v, let $\mathrm{L}(v)$ and $\mathrm{R}(v)$ denote its left and right child, respectively.

4.1 Maintaining $\mathbb{T}(\mathbb{P})$

For a node $v \in \mathbb{T}(\mathbb{P})$, let $\mathbb{P}_v \subseteq \mathbb{P}$ be the set of pillars whose x-projections contain δ_v but not $\delta_{p(v)}$. Let $\mathbb{P}^*_v = \bigcup_w \mathbb{P}_w$, over all descendents w of v (including v itself), and let $\pi(v)$ be the length of the portion of δ_v covered by the x-projections of the pillars in $\mathbb{P}^*_v$. At each node $v \in \mathbb{T}(\mathbb{P})$ we maintain $|\mathbb{P}_v|$ and $\pi(v)$. For a leaf v, $\pi(v) = \|\delta(v)\|$ if $\mathbb{P}_v \neq \emptyset$ and $\pi(v) = 0$ otherwise. If v is an interior node then

$$\pi(v) \quad = \quad \begin{cases} \|\delta_v\| & \text{if } \mathbb{P}_v \neq \emptyset, \\ \pi(\mathrm{L}(v)) + \pi(\mathrm{R}(v)) & \text{otherwise.} \end{cases} \qquad (3)$$

Clearly, if u is the root of $\mathbb{T}(\mathbb{P})$ then $\pi(u)$ is the value of π, i.e., the length of the portion of the top edge of $\square$ (or any other horizontal line intersecting $\square$) covered by the pillars.

Inserting/deleting a pillar. Let P be a new pillar that we want to insert into $\mathbb{T}(\mathbb{P})$. For each node $v \in \mathcal{C}(I_P)$, we increment $\|\mathbb{P}_v\|$ and set $\pi(v) = \|\delta_v\|$ (since $P \in \mathbb{P}_v$). Then we proceed, bottom-up, through the nodes of $\mathcal{N}(I_P)$, and, for each such node v, we update $\pi(v)$ using (3). Deletion of a pillar from $\mathbb{T}(\mathbb{P})$ is done symmetrically. Each of these operations takes $O(\log n)$ time.

Answering pillar-length queries. A *pillar-length* query, with an interval $I \subseteq [x_0, x_1]$, seeks the value of $\pi(I)$, the length of the portion of I covered by the x-projections of the pillars. The recursive

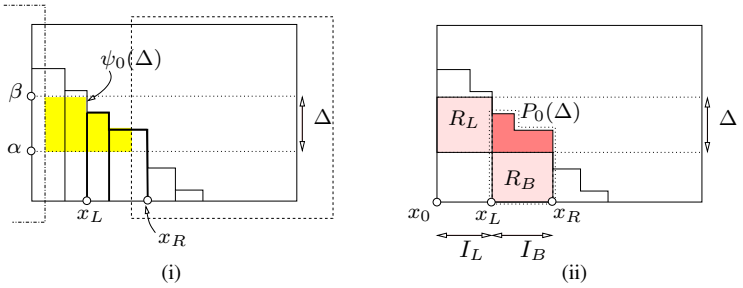

Figure 7. (i) Shaded region is $\psi_0(\Delta)$, rectangles in $\tilde{\mathbb{C}}_0(\Delta)$ are drawn by thick lines; pillars are drawn by dashed lines. (ii) Rectangles R_B and R_L; dotted lines surround $P_0(\Delta)$.

procedure described in Figure 6 computes $\pi(I)$. It is a standard 1-dimensional range-searching procedure, with one caveat: the endpoints of I may lie in the interior of the intervals associated with the corresponding leaves of $\mathbb{T}(\mathbb{P})$, so additional (though obvious) actions are required at those leaves. The correctness is straightforward, and the running time is obviously $O(\log n)$. Pillar-length queries will be used in Section 4.2.

4.2 Maintaining $\mathbb{T}(\tilde{\mathbb{C}})$

We next describe the information stored at the nodes of $\mathbb{T}(\tilde{\mathbb{C}})$ and its update and query procedures. For each node $v \in \mathbb{T}(\tilde{\mathbb{C}})$ and for $i = 0, 1, 2, 3$, we define $\tilde{\mathbb{C}}_{iv}$ to be the subset of rectangles C in $\tilde{\mathbb{C}}_i$ such that $\delta_v \subseteq I_C \subset \delta_{p(v)}$; set $\tilde{\mathbb{C}}^*_{iv} = \bigcup_w \tilde{\mathbb{C}}_{iw}$, over all descendents w of v (including v itself). Since the rectangles in $\tilde{\mathbb{C}}_i$ are pairwise disjoint and a vertical line intersects at most one of them, $\tilde{\mathbb{C}}_{iv} \neq \emptyset$ implies that $\tilde{\mathbb{C}}^*_{iv} = \tilde{\mathbb{C}}_{iv}$ is a singleton set $\{C_{iv}\}$. At each node v of $\mathbb{T}(\tilde{\mathbb{C}})$, for $i = 0, 1, 2, 3$, we maintain the rectangle C_{iv} if $\tilde{\mathbb{C}}_{iv} \neq \emptyset$ and the following two pieces of information:

$\xi_i(v)$: $\text{Area}([\mathcal{U}(\tilde{\mathbb{C}}^*_{iv}) \setminus \mathcal{U}(\mathbb{P}^*_v)] \cap \square_v)$, for $i = 0, \ldots, 3$.

$J_i(v)$: The smallest vertical segment that contains the y-coordinates of all the horizontal edges of the staircase polygon P_i that cross $\square_v$. If $\square_v$ intersects only one (resp., no) horizontal edge of the chain, then $J_i(v)$ is a singleton (resp., empty). The segments $J_i(v)$ at the leaves are ordered in the (increasing or decreasing) y-direction because $P_i(v)$ is a staircase, and therefore monotone, polygon.

As already noted, if $\tilde{\mathbb{C}}_{iv} \neq \emptyset$, then it consists of a single rectangle C_{iv}. We denote by χ_{iv} the y-coordinate of the horizontal edge of C_{iv} that lies in the interior of $\square_v$. Hence, for an interior node v we have that

$$\xi_i(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ (\|\delta_v\| - \pi(v))\,\text{Ht}(C_{iv}) & \text{if } \mathbb{P}_v = \emptyset,\ \tilde{\mathbb{C}}_{iv} \neq \emptyset, \\ \xi_i(\text{L}(v)) + \xi_i(\text{R}(v)) & \text{otherwise.} \end{cases} \quad (4)$$

$$J_i(v) = \begin{cases} [\chi_{iv}, \chi_{iv}] & \text{if } \tilde{\mathbb{C}}_{iv} \neq \emptyset, \\ \text{conv}(J_i(\text{L}(v)) \cup J_i(\text{R}(v))) & \text{otherwise.} \end{cases} \quad (5)$$

Here $\text{conv}(X)$ is the smallest interval containing X, and $\text{Ht}(\varrho)$ denotes the height of a rectangle ϱ. If v is a leaf, a similar equation holds, without the recursive terms involving the children.

Inserting/deleting a pillar. Since $\xi_i(v)$ depends on $\pi(v)$, we update $\mathbb{T}(\tilde{\mathbb{C}})$ when a pillar is inserted or deleted, even though pillars are not explicitly stored in $\mathbb{T}(\tilde{\mathbb{C}})$. Suppose we insert or delete a pillar P. After updating $\mathbb{T}(\mathbb{P})$, we update $\xi_i(v)$ and $J_i(v)$ at various nodes, as follows. For each node $v \in \mathcal{C}(I_P)$ and for each $i = 0, 1, 2, 3$, we set $\xi_i(v) = 0$ (since $\pi(v) = \|\delta_v\|$). Then we proceed through the nodes of $\mathcal{N}(I_P)$ in a bottom-up fashion, and update $\xi_i(v)$ at each such node v using (4).

Inserting/deleting a corner. To insert a rectangle $C \in \tilde{\mathbb{C}}_i$, we first store C as the rectangle C_{iv} for all nodes $v \in \mathcal{C}(I_C)$. Then we proceed through the nodes of $\mathcal{C}(I_C) \cup \mathcal{N}(I_C)$ in a bottom-up fashion, and, for each such node v, we update $\xi_i(v)$ and $J_i(v)$ according to (4) and (5), respectively. Since $\tilde{\mathbb{C}}_{iv} \neq \emptyset$ for $v \in \mathcal{C}(I)$, updating $\xi_i(v)$ and $J_i(v)$ for $v \in \mathcal{C}(I_C)$ does not require the recursive terms of (4) and (5).

Answering a corner-area query. Let $\Delta = [\alpha, \beta] \subseteq [y_0, y_1]$ be a y-interval. We describe how to compute

$$\psi(\Delta) = \text{Area}[\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta.$$

Clearly, $\psi(\Delta) = \sum_{i=0}^{3} \psi_i(\Delta)$, where $\psi_i(\Delta) = \text{Area}[\mathcal{U}(\tilde{\mathbb{C}}_i) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta$; see Figure 7 (i). We describe how to compute $\psi_0(\Delta)$; the other $\psi_i(\Delta)$'s are computed in a similar manner.

Let $\tilde{\mathbb{C}}_0(\Delta) \subseteq \tilde{\mathbb{C}}_0$ be the set of rectangles whose top edges have y-coordinates in Δ; see Figure 7 (ii). The rectangles in $\tilde{\mathbb{C}}_0(\Delta)$ form a contiguous subsequence of $\tilde{\mathbb{C}}_0$, and $\mathcal{U}(\tilde{\mathbb{C}}_0(\Delta))$ is also a staircase polygon $P_0(\Delta)$. Let x_L (resp., x_R) be the x-coordinate of the left (resp., right) boundary of $P_0(\Delta)$. Set $I_L = [x_0, x_L]$, $I_B = [x_L, x_R]$, $R_L = I_L \times \Delta$, and $R_B = I_B \times [y_0, \alpha]$ (see Figure 7 (ii)). Then

$$\mathcal{U}(\tilde{\mathbb{C}}_0) \cap W_\Delta = [P_0(\Delta) \setminus R_B] \cup R_L.$$

Since R_L and $P_0(\Delta)$ are disjoint and $R_B \subseteq P_0(\Delta)$, we have

$$\begin{aligned} \psi_0(\Delta) &= \text{Area}(P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})) - \text{Area}(R_B \setminus \mathcal{U}(\mathbb{P})) \\ &\quad + \text{Area}(R_L \setminus \mathcal{U}(\mathbb{P})) \\ &= \text{Area}(P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})) - (x_R - x_L - \pi(I_B))(\alpha - y_0) \\ &\quad + (x_L - x_0 - \pi(I_L))(\beta - \alpha). \end{aligned}$$

We compute $\pi(I_B)$ and $\pi(I_L)$ by performing two pillar-length queries on $\mathbb{T}(\mathbb{P})$ with I_B and I_L, respectively. We then compute $P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})$ by invoking the recursive procedure described in Figure 8 with the root of $\mathbb{T}(\mathbb{C})$ and Δ.

The running time of the procedure PSI-QUERY is $O(\log n)$ because the intervals $J_0(v)$ are ordered along the leaves of $\mathbb{T}(\tilde{\mathbb{C}})$ and the parents of the nodes visited by the procedure lie on two paths of $\mathbb{T}(\tilde{\mathbb{C}})$.

4.3 Maintaining $\mathbb{T}(\mathcal{R})$

For each $v \in \mathbb{T}(\mathcal{R})$, we define $\mathcal{R}_v \subseteq \mathcal{R}$ to be the subset of rectangles of $\mathcal{R}$ whose x-projections contain δ_v but not $\delta_{p(v)}$, and set $\mathcal{R}^*_v = \bigcup_w \mathcal{R}_w$, over all descendents w of v (including v itself).

```
PSI-QUERY(v, Δ)
  if Δ ∩ J_0(v) = ∅   return 0
  if J_0(v) ⊆ Δ       return ξ_0(v)
  return PSI-QUERY(L(v), Δ)
         + PSI-QUERY(R(v), Δ)
```

Figure 8. The recursive procedure to compute $P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})$.

At each node v of $\mathbb{T}(\mathcal{R})$, we store the set $\mathcal{R}_v$ and the following auxiliary information:

$\varphi(v)$: Area of $\mathcal{U}(\mathcal{R}_v^*) \cap \square_v$ not covered by the long squares (pillars, upper rim, lower rim) of $\mathcal{S}$.

$h(v)$: Length of the left edge of $\square_v$ covered by the rectangles in $\mathcal{R}_v$ but not by the squares in the upper or lower rim.

$\lambda_f(v)$: Length of the floor covered by the x-projections of the stalagmites in $\mathcal{R}_v^*$ but not by the pillars.

$\lambda_c(v)$: Length of the ceiling covered by the x-projections of the stalactites in $\mathcal{R}_v^*$ but not by the pillars.

$\mathrm{fl}(v)$: The most recently recorded value of fl at v.

$\mathrm{cl}(v)$: The most recently recorded value of cl at v.

We remark right away that not all these values are correctly maintained at all times at v, but they are maintained in such a way that it is easy to reset them (in $O(1)$ time) to the correct values upon demand—see below.

The values stored at the root of $\mathbb{T}(\mathcal{R})$ immediately give the values of φ that we maintain for $\square$, as required by the algorithm described in Section 3. Let ϱ_v be the rectangle $\delta_v \times [\mathrm{fl}, \mathrm{cl}]$, assuming $\mathrm{fl} \le \mathrm{cl}$. Since, by construction, every rectangle in $\mathcal{R}_v$ extends from the left edge to the right edge of $\square_v$, we have

$$h(v) = \sum_{R \in \mathcal{R}_v} \mathrm{Ht}(R \cap \varrho_v). \tag{6}$$

Note that $h(v)$ depends only on $\mathcal{R}_v$ and not on the other rectangles in the sets $\mathcal{R}_w$, for *proper* descendants w of v. Moreover, there is at most one rectangle in R_v that intersects the floor (resp., ceiling). Hence, $\mathrm{Ht}(R \cap \varrho_v)$ is either 0 or $\mathrm{Ht}(R)$ except for at most two rectangles in $\mathcal{R}_v$.

Let w and z be the children of an interior node v. The following equalities are obvious (see Figure 9):

$$\varphi(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \varphi(w) + \varphi(z) + (\|\delta_v\| - \pi(v))h(v) & \text{otherwise.} \end{cases} \tag{7}$$

$$\lambda_f(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \|\delta_v\| & \text{if } \mathbb{P}_v = \emptyset,\ \mathcal{R}_v \cap \mathbb{S}\mathrm{g} \neq \emptyset, \\ \lambda_f(w) + \lambda_f(z) & \text{otherwise.} \end{cases} \tag{8}$$

$$\lambda_c(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \|\delta_v\| & \text{if } \mathbb{P}_v = \emptyset,\ \mathcal{R}_v \cap \mathbb{S}\mathrm{c} \neq \emptyset, \\ \lambda_c(w) + \lambda_c(z) & \text{otherwise.} \end{cases} \tag{9}$$

We can use simpler variants of the above equations, which do not involve the recursive terms, for the values of these quantities at the leaf nodes; for example, $\varphi(v)$ is 0 if $\mathbb{P}_v \neq \emptyset$, and is $\|\delta_v\| \cdot h(v)$ otherwise.

$\mathbb{T}(\mathcal{R})$ always maintains the correct values of $\lambda_c(v)$ and $\lambda_f(v)$ at all nodes, but the values of $\mathrm{fl}(v), \mathrm{cl}(v), \varphi(v), h(v)$ may be incorrect because the updating of the floor and ceiling of $\square$ does not always reach all the nodes of $\mathbb{T}(\mathcal{R})$ in "real time". For example, if the floor is raised when we insert a square of the lower rim, so that no edge in Λ lies between the new and old floor, then we do not update $\mathbb{T}(\mathcal{R})$ at all, and thus $\mathrm{fl}(v), \mathrm{cl}(v), \varphi(v)$, and $h(v)$ are not modified, even though they have changed. Even when $\mathbb{T}(\mathcal{R})$ gets updated, when the floor or ceiling sweeps through a horizontal floater edge, not all the nodes of $\mathbb{T}(\mathcal{R})$ "get the news"—it is too expensive to broadcast the changes explicitly to all nodes of $\mathbb{T}(\mathcal{R})$. Instead, we update them in a lazy manner, so that the following two invariants are maintained:

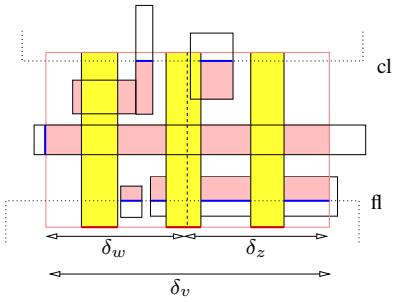

Figure 9. Illustrating the data stored at a node v of the various segment trees: $\pi(v)$ is the highlighted portion of the bottom edge; $\lambda_f(v)$ is highlighted on the floor, and $\lambda_c(v)$ is highlighted on the ceiling; $h(v)$ is highlighted on the left edge, and $\varphi(v)$ is the darker region.

(I1) *For any node $v \in \mathbb{T}(\mathcal{R})$, none of the y-coordinates of the horizontal edges of rectangles in $\mathcal{R}_v^*$ lie between* fl *and* $\mathrm{fl}(v)$ *or between* cl *and* $\mathrm{cl}(v)$.

(I2) The value of $\varphi(v)$ *gives the area of $\mathcal{U}(\mathcal{R}_v^*) \cap \square_v$ not covered by the long squares of $\mathcal{S}$, under the assumption that the floor (resp., ceiling) is at* $\mathrm{fl}(v)$ *(resp.,* $\mathrm{cl}(v)$*).*

To ensure that these quantities at a node v are correct whenever we access v (either to update the information stored at v or to recompute the information at $p(v)$), we apply the two straightforward subroutines described in Figure 10, before manipulating/accessing any other data at v.

By invariant (I1), there is no horizontal edge of $\mathcal{R}_v^*$ between fl and $\mathrm{fl}(v)$, so if we raise the floor from $\mathrm{fl}(v)$ to fl then the value of $\varphi(v)$ decreases by the amount $\lambda_f(v)[\mathrm{fl} - \mathrm{fl}(v)]$. A similar argument justifies the updating applied to $h(v)$, and for the ceiling. This, in conjunction with (I2), implies that, after executing these procedures, $\mathrm{fl}(v)$, $\mathrm{cl}(v)$, $\varphi(v)$ and $h(v)$ have their correct values.

Whenever we want to compute the value of φ in Section 3, after inserting or deleting a square in $\mathcal{S}$, we call ADJUSTFLOOR and ADJUSTCEILING at the root u in $\mathbb{T}(\mathbb{F})$ and return the value of $\varphi(u)$.

Inserting/deleting a pillar. Although pillars are not stored explicitly in $\mathbb{T}(\mathcal{R})$, as for $\mathbb{T}(\tilde{\mathbb{C}})$, we update the auxiliary information in $\mathbb{T}(\mathcal{R})$ whenever a pillar is inserted or deleted to reset the quantities $\pi(v)$. Suppose we insert or delete a pillar P. We first visit the nodes v in $\mathcal{N}(I_P)$ in a top-down manner and call the functions ADJUSTFLOOR and ADJUSTCEILING at v, and at its children if v is an interior node. Next, we update $\varphi(v), \lambda_f(v), \lambda_c(v), \mathrm{fl}(v)$, and $\mathrm{cl}(v)$ at all nodes $v \in \mathcal{N}(I_P) \cup \mathcal{C}(I_P)$ in a bottom-up manner using (7)–(9). The total time spent is $O(\log n)$.

Inserting/deleting a floater. Suppose we want to insert or delete a rectangle $R \in \mathcal{R}$. As for the case of pillars, we first visit the nodes v in $\mathcal{N}(I_R)$ in a top-down manner and call the functions

```
ADJUSTFLOOR(v)
    φ(v) = φ(v) − λ_f(v)[fl − fl(v)]
    if R(v) ∩ Sg ≠ ∅
        h(v) = h(v) − [fl − fl(v)]
    fl(v) = fl
```

```
ADJUSTCEILING(v)
    φ(v) = φ(v) − λ_c(v)[cl(v) − cl]
    if R(v) ∩ Sc ≠ ∅
        h(v) = h(v) − [cl(v) − cl]
    cl(v) = cl
```

Figure 10. Subroutines to adjust the information at the nodes of $\mathbb{T}(\mathbb{F})$.

ADJUSTFLOOR and ADJUSTCEILING at v, and at its children if v is an interior node. Next, we perform the following step at each node $v \in \mathcal{C}(I_R)$, i.e., at those nodes for which $R \in \mathcal{R}_v$. Let $\varrho_v = \delta_v \times [\mathrm{fl}, \mathrm{cl}]$ and $r = \mathrm{Ht}(R \cap \varrho_v)$. If we are inserting R, we set $h(v) = h(v) + r$, $\varphi(v) = \varphi(v) + (\|\delta_v\| - \pi(v))r$, and set $\lambda_f(v)$ (resp., $\lambda_c(v)$) to $\|\delta_v\| - \pi(v)$ provided R is a rectangle of $\mathbb{S}\mathrm{g}$ (resp., $\mathbb{S}\mathrm{c}$). (Note that the values of $h(v)$, $\varphi(v)$ in the right-hand sides of the equations are correct because the ADJUSTFLOOR and ADJUSTCEILING subroutines have just been called at v.) On the other hand, if we are deleting R, we set $h(v) = h(v) - r$, $\varphi(v) = \varphi(v) - (\|\delta_v\| - \pi(v))r$, and set $\lambda_f(v)$ (resp., $\lambda_c(v)$) to zero, provided R is a rectangle of $\mathbb{S}\mathrm{g}$ (resp., $\mathbb{S}\mathrm{c}$). The last action is justified by noting that if $R \in \mathbb{S}\mathrm{g}$ then no other rectangle of $\mathcal{R}_v^*$ belongs to $\mathbb{S}\mathrm{g}$, and the same is true for $\mathbb{S}\mathrm{c}$.

Finally, we update $\varphi(v), \lambda_f(v), \lambda_c(v), \mathrm{fl}(v)$, and $\mathrm{cl}(v)$ at all nodes $v \in \mathcal{N}(I_R) \cup \mathcal{C}(I_R)$ in a bottom-up manner using (7)–(9). The total time spent is $O(\log n)$.

Putting everything together, we obtain the following lemma which summarizes the properties of our segment trees.

LEMMA 4.1. *Each of the following operations can be performed in $O(\log n)$ time:*

(i) insertion or deletion of a pillar, including the updates in $\mathbb{T}(\tilde{\mathbb{C}})$ and $\mathbb{T}(\mathcal{R})$;

(ii) insertion or deletion of a corner or a floater rectangle;

(iii) answering a pillar-length or a corner-area query.

5. CONCLUSIONS

In this paper we described an $O(n^{4/3} \log n)$-time algorithm for computing the volume of the union of a set of cubes in $\mathbb{R}^3$. A natural question is whether the running time can be improved to $O(n \operatorname{polylog}(n))$. As remarked in the introduction, we believe that our approach can be extended to get such an improved algorithm but so far we have not been able to circumvent various technical difficulties. Our algorithm heavily uses the fact that the input boxes are cubes, so a faster algorithm for computing the union of general (axis-aligned) boxes in $\mathbb{R}^3$ remains elusive.

References

[1] http://en.wikipedia.org/wiki/Klee's_measure_problem

[2] D. Attali and H. Edelsbrunner, Inclusion-exclusion formulas from independent complexes, *Proc. 21st Ann. Sympos. Comput. Geom.*, 2005, 247–254.

[3] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd edition, Springer Verlag, Heidelberg, 2000.

[4] J. L. Bentley, Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.

[5] J.D. Boissonnat, M. Sharir, B. Tagansky and M. Yvinec, Voronoi diagrams in higher dimensions under certain polyhedral distance functions, *Discrete Comput. Geom.* 19 (1998), 485–519.

[6] E. Chen and T. M. Chan, Space-efficient algorithms for Klee's measure problem, *Proc. 17th Canadian Conf. Comput. Geom.*, 2005.

[7] B. S. Chlebus, On the Klee's measure problem in small dimensions, *Proc. 25th Conf. Current Trends in Theory and Practice of Informatics*, 1998, 304-311.

[8] H. Edelsbrunner, The union of balls and its dual shape, *Discrete Comput. Geom.* 13 (1995), 415–440.

[9] M. L. Fredman and B. Weide, The complexity of computing the measure of $\bigcup[a_i, b_i]$, *Commun. ACM* 21 (1978), 540–544.

[10] J. van Leeuwen and D. Wood, The measure problem for rectangular ranges in *d*-space, *J. Algorithms* 2 (1981), 282–300.

[11] V. Klee, Can the measure of $\bigcup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *Amer. Math. Monthly* 84 (1977), 284–285.

[12] M. Overmars and C.K. Yap, New upper bounds in Klee's measure problem, *SIAM J. Comput.* 20 (1991), 1034–1045.