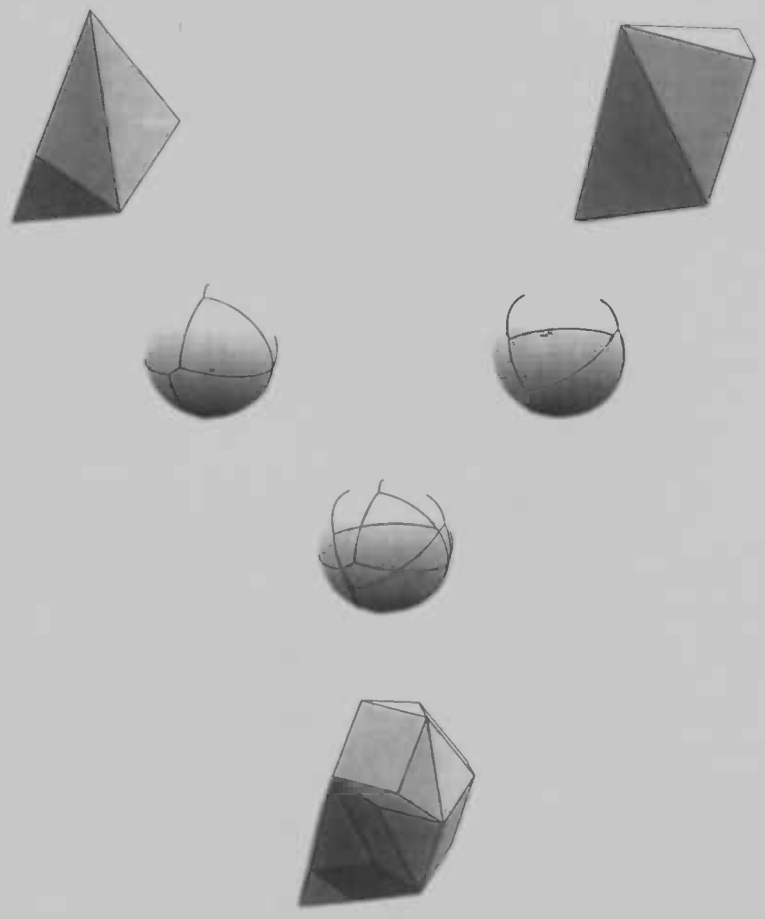
WORDT
NIET UITGELEEND

Calculating the Minkowski sum of convex 3D polyhedra using a sphere sweep algorithm and attributed graphs



G. Jorna
Thesis advisor: H. Bekker

Introduction

Let A and B be two convex polyhedra in Euclidean n-dimensional space $\mathbb{R}^n$ with nv_A and nv_B vertices respectively. The Minkowski sum of A and B is the convex polyhedron

$$C = A \oplus B = \{a + b \mid a \in A, b \in B\}.$$

In this thesis we are only interested in convex polyhedra in three-dimensional space $\mathbb{R}^3$. In the rest of this thesis we will omit the words 'convex' and 'three-dimensional', so when we say polyhedron we mean convex polyhedron in three-dimensional space, unless stated otherwise.

Several methods are known to calculate the Minkowski sum. We will mention some of them here.

The first method to calculate the Minkowski sum C of polyhedra A and B is very trivial, yet highly time consuming.

First all vertex positions of B are added to all vertex positions of A, resulting in a set of $nv_A \times nv_B$ points. Calculating the convex hull of this pointset gives C.

Other, more efficient methods are based on the use of *slope diagrams*, that is, diagrams on the unit sphere S^2. The overlay of the slope diagrams SDA and SDB of A and B is calculated. For every face in the overlay we have to determine in which faces of SDA and SDB it is located. This is called *face location*.

Bekker and Roerdink[1] proposed a method that is also based on the overlay of slope diagrams, but avoids face location. A slope diagram SDC of the still unknown polyhedron C contains nearly all necessary information to construct polyhedron C. Only the dimensions and the position of C can not be determined from SDC. Bekker and Roerdink store additional attributes with SDC. These attributes provide the information needed to construct C and shift it to the proper position. Face location is no longer necessary. In this thesis we will present this method and give an implementation and results of this implementation.

Several methods are known for the calculation of the overlay of two slope diagrams. A more detailed explanation of slope diagrams will follow later. For now we only say that a slope diagram is a spherical graph positioned on the unit sphere.

In his thesis De Raedt [4] presents an algorithm to calculate the overlay of connected subdivisions on a sphere using a planar overlay algorithm which runs in $O(N \log N)$ time. A similar method was presented by Granados et al. [7]. These algorithms are based on overlay algorithms in the plane. In order to calculate the overlay in the plane a switch is to be made from S^2

to $\mathbb{R}^2$ space. They do this by projecting the slope diagrams on a plane and calculate the overlay in $\mathbb{R}^2$ space. Then they project the overlay back on the unit sphere. Fogel and Halperin [6] also used an overlay algorithm in the plane. They used a projection of the slope diagram on the unit cube.

We want an algorithm that avoids the switch between S^2 to $\mathbb{R}^2$ space. The overlay of the slope diagrams should be calculated directly on the unit sphere. As a basis for our 3D overlay algorithm we will use a planar overlay algorithm.

In [2] Berg et al. described the *plane sweep algorithm*, an algorithm that computes the intersection points of n segments in the plane in $O(n \log n)$ time. In section 2 we will give a detailed explanation of this algorithm.

We will design our 3D algorithm to work analogue to the plane sweep algorithm, but there are some differences. Working with 3D vertices will force us to use different orderings on vertices and segments. Intersection of segments is defined differently. We have to choose another form of sweep line with another sort of sweep movement.

The reason that we want to develop a new overlay algorithm that computes the overlay of two slope diagrams is that we want to avoid the hassle of face location, and we want to store and transfer edge information.

Summarizing, the algorithm we implement and test avoids face location. For this algorithm two new techniques are used. A sweep algorithm is used to calculate the overlay of slope diagrams on the unit sphere. Attributed slope diagrams are used to store positional information.

The objective of our work is to design an efficient algorithm to calculate the Minkowski sum of two three-dimensional convex polyhedra, implement the algorithm, and run tests to analyze whether the algorithm works and is indeed more efficient than existing methods for calculation of the Minkowski sum.

1 Preliminaries

As mentioned in the introduction we can calculate the Minkowski sum of polyhedra in n-dimensional space $\mathbb{R}^n$. Later in this thesis we will only speak about polyhedra in three-dimensional space $\mathbb{R}^3$. But first we will give a simple example of a Minkowski sum of two polygons in two-dimensional space $\mathbb{R}^2$.

1.1 Minkowski sum

Suppose we have a square s (Figure 1a) and a triangle t (Figure 1b), and we want to draw the Minkowski sum ms of s and t. We do this as follows. We pick a point p that lies within t. Assume t to be a triangular shaped brush. To create the Minkowski sum we place t over s so that p lies within the boundaries of s. Now we start moving t around anywhere we can without letting p cross the boundaries of s. The *brushed* area forms the Minkowski sum ms.

Figure 1c shows ms. The Minkowski sum itself is drawn as the thick black line. The original square and triangle are also displayed, whereas the dashed lines display the triangle in the positions where p coincides with the vertices of the square.

As we can see all edges of s and t return in ms. Edge $e_{s,1}$ returns as $e_{ms,1}$, $e_{s,2}$ returns as $e_{ms,3}$, and so on. Edge $e_{t,1}$ returns as $e_{ms,2}$, $e_{t,2}$ returns as $e_{ms,4}$, and so on.

Less obvious is what happened to the parallel edges $e_{s,4}$ and $e_{t,3}$. These edges added up to an edge $e_{ms,6}$ that is parallel as well and has a length that is the sum of the lengths of $e_{s,4}$ and $e_{t,3}$.

It can be easily seen that the shape of ms is independent of the point within t that we choose for p. The position however is dependent of this choice. We will come back to this later.

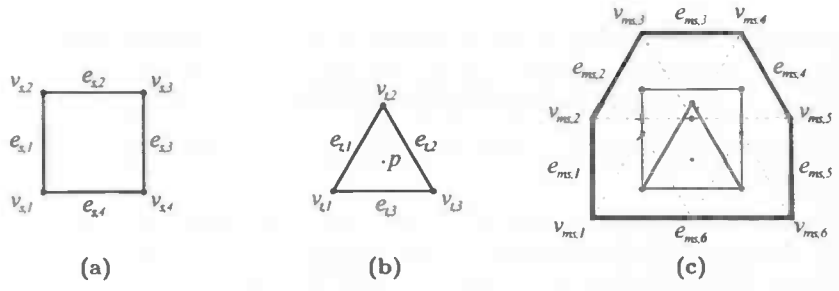


Figure 1: *Example of a Minkowski sum in two-dimensional space $\mathbb{R}^2$. (c) shows the Minkowski sum of a square (a), and a triangle (b).*

In a more mathematical way, we can explain the Minkowski sum as follows. We take a triangle t and a square s. We superimpose the vertices of t (Figure 2b) over the vertices of s (Figure 2a).

We take a point p as origin and calculate the vertex positions of the vertices of both s and t with respect to p. We now add all vertex positions of s to all vertex positions of t. In Figure 2c the vertex positions of vertices $v_{s,2}$ and $v_{t,2}$ add up to vertex position v_{ms}. Likewise we can add up the vertex positions of the other vertices. The result is a set M of at most $nv_s \times nv_t$ new vertex positions, nv_s is the number vertices is s, nv_t the number of vertices in t. At most, since vertex positions might coincide.

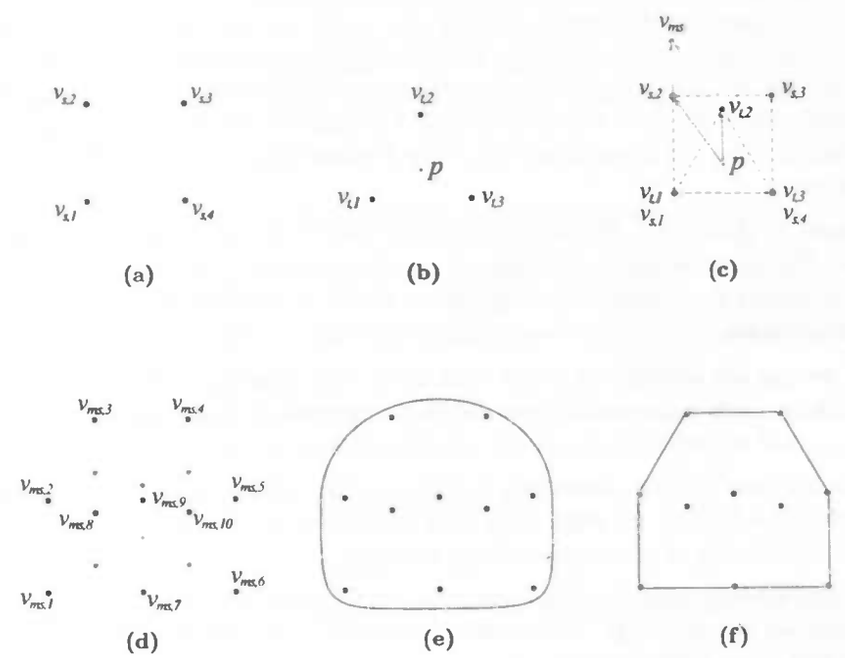


Figure 2: *The Minkowski sum in a more mathematical picture.*

In Figure 2d all vertices of M are drawn in black, the vertices and edges of s en t are drawn in gray. Although we expected $4 \times 3 = 12$ vertices in M, there are only 10. This is obviously caused by coinciding vertices. For example, the vertex positions of $v_{t,1}$ and $v_{s,3}$ add up to the same vertex position as $v_{t,3}$ and $v_{s,2}$.

We get the Minkowski sum of s and t by calculating the convex hull of M. Imagine the vertices are needles pinned down on a plank. We put an elastic thread around the needles, so that all needles are positioned inside the thread (Figure 2e). When the thread is released it will wrap around the outer most needles (Figure 2f).

Back to the vertices again. The line around the vertices is the convex hull. That is, only the vertices that influence the shape of the convex hull are actually part of it. The vertices that do not touch the line can be deleted from M. Furthermore, the vertices that are on the line, but can be deleted without changing the shape of the line, do not belong to the convex hull either. In Figure 2f we see the same shape as in Figure 1c. It is the shape of the Minkowski sum of s and t.

So far we have only spoken about the shape of the Minkowski sum. Its position, obviously, is dependent on the positions of s and t. In Figures 1 and 2 we computed the Minkowski sum as if p was the point $(0,0)$ and inside both s and t.

In Figure 3 we drew the triangle, square, and their Minkowski sum in the right positions. Vertex positions were computed with respect to the origin $(0,0)$.

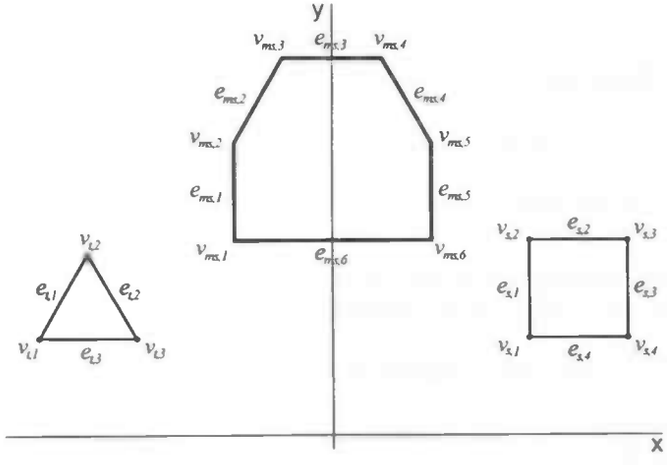


Figure 3: *The Minkowski sum of s and t in the right position.*

In three-dimensional space $\mathbb{R}^3$ the Minkowski sum works analogue to the previous example.

1.2 Methods to calculate the Minkowski sum

Several methods are known to calculate the Minkowski sum. We will explain some of them here.

1.2.1 Method 1 – Naive method

The first method to calculate the Minkowski sum is a very naive method. It is simple, but time consuming. In fact, it is the method we used to explain

the notion Minkowski sum in section 1.1. Assume we want to calculate the Minkowski sum of two polyhedra A and B with n vertices each. We add all n vertex positions of A to all n vertex positions of B. The result is a set of n^2 vertex positions. Of this set we have to calculate the convex hull. The time complexity of the calculation of the convex hull is $O(m \log m)$ where m is the number of vertices. Our set has n^2 vertices, so calculating the convex hull would require $O(n^2 \log n^2)$ time! A very time consuming process indeed.

This method is inefficient, because in general many of the n^2 vertices in the intermediary set are not part of the convex hull, either because they lie in the interior of the convex hull, or because several vertices coincide.

In order to design a more efficient method to calculate the Minkowski sum it is important to avoid adding up vertices from A and B that will not be part of the Minkowski sum. This is possible by using slope diagram representation of polyhedra. Slope diagrams are based on *support functions* en *support sets*.

Support function In 3D space the *support function* is defined by

$$h(A, u) = \sup\{\langle a, u\rangle \mid a \in A\}, \qquad u \in S^2.$$

Here $\langle a, u\rangle$ is the inner product of vectors a and u, and S^2 denotes the unit sphere.

We will now explain the support function for a polygon in two-dimensional space. We define the support function in 2D space as follows.

$$h(B, u) = \sup\{\langle a, u\rangle \mid a \in B\}, \qquad u \in S^1,$$

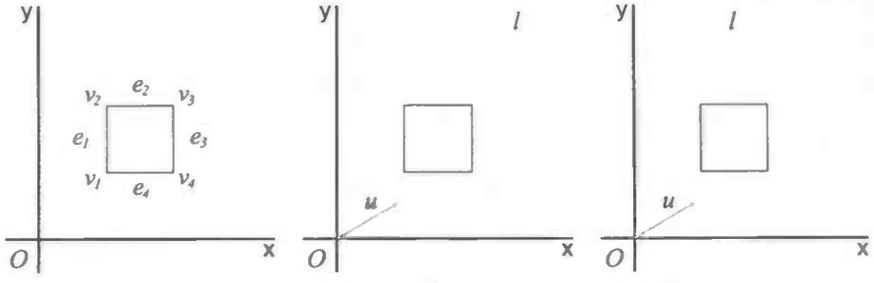
where S^1 denotes the unit circle.

In words, $h(B, u)$ is the supremum of the set of inner products of u with all points on polygon B.

In a more graphical way we could describe the support function as follows. Let B be a square with vertices $v_1 = (1, 1)$, $v_2 = (1, 2)$, $v_3 = (2, 2)$, and $v_4 = (2, 1)$ (Figure 4a). Let u be the unit vector $(\frac{1}{2}\sqrt{3}, \frac{1}{2})$. Let line l be a line perpendicular to u and infinitely far away from the origin O (Figure 4b). Now we move l closer to O in the direction $-u$ until it bounces against B. This happens when l reaches vertex v_3 (Figure 4c).

Since u is the unit vector $h(B, u)$ now is the distance of l to the origin. In this particular case $h(B, u) = \langle v_3, u\rangle = \langle (2, 2), (\frac{1}{2}\sqrt{3}, \frac{1}{2})\rangle = 1 + \sqrt{3}$. For any other point a on B inner product $\langle a, u\rangle < 1 + \sqrt{3}$.

The *support set* $F(B, u)$ of B at $u \in S^1$ consists of all points $a \in B$ for which $\langle a, u\rangle = h(B, u)$.



(a) *Square s.*

(b) *A randomly chosen unit vector u and line l perpendicular to u.*

(c) *l bounces against s. $h(B,u)$ is the distance between O and l.*

(d) *Another orientation for u, another value for $h(B,u)$, but the same support set $F(B,u)$.*

(e) *Yet another orientation of u. $F(B,u)$ are all points on edge e_2.*

(f) *The slope diagram of s.*

Figure 4: *The support function in graphics.*

Let us take a look at Table 1. We have computed the inner products of nine unit vectors $u_1, \ldots, u_9$ with the four vertices of square B, and determined the support function of each unit vector. It is easy to see that the inner products of the points on an edge e of B with a unit vector u_i are directly related to the inner products of the endpoints of e with u_i.

i	u_i	$\langle v_1, u_i \rangle$	$\langle v_2, u_i \rangle$	$\langle v_3, u_i \rangle$	$\langle v_4, u_i \rangle$	$h(B, u_i)$
1	$(1,0)$	1	1	$\mathbf{2}$	$\mathbf{2}$	$\mathbf{2}$
2	$(\frac{1}{2}\sqrt{3}, \frac{1}{2})$	$\frac{1}{2}\sqrt{3}+\frac{1}{2}$	$\frac{1}{2}\sqrt{3}+1$	$\mathbf{\sqrt{3}+1}$	$\sqrt{3}+\frac{1}{2}$	$\mathbf{\sqrt{3}+1}$
3	$(\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2})$	$\sqrt{2}$	$1\frac{1}{2}\sqrt{2}$	$\mathbf{2\sqrt{2}}$	$1\frac{1}{2}\sqrt{2}$	$\mathbf{2\sqrt{2}}$
4	$(\frac{1}{2}, \frac{1}{2}\sqrt{3})$	$\frac{1}{2}\sqrt{3}+\frac{1}{2}$	$\sqrt{3}+\frac{1}{2}$	$\mathbf{\sqrt{3}+1}$	$\frac{1}{2}\sqrt{3}+1$	$\mathbf{\sqrt{3}+1}$
5	$(0,1)$	1	$\mathbf{2}$	$\mathbf{2}$	1	$\mathbf{2}$
6	$(-\frac{1}{2}, \frac{1}{2}\sqrt{3})$	$\frac{1}{2}\sqrt{3}-\frac{1}{2}$	$\mathbf{\sqrt{3}-\frac{1}{2}}$	$\sqrt{3}-1$	$\frac{1}{2}\sqrt{3}-1$	$\mathbf{\sqrt{3}-\frac{1}{2}}$
7	$(\frac{1}{2}\sqrt{2}, \frac{1}{2}\sqrt{2})$	0	$\mathbf{\frac{1}{2}\sqrt{2}}$	0	$-\frac{1}{2}\sqrt{2}$	$\mathbf{\frac{1}{2}\sqrt{2}}$
8	$(-\frac{1}{2}\sqrt{3}, \frac{1}{2})$	$-\frac{1}{2}\sqrt{3}+\frac{1}{2}$	$\mathbf{-\frac{1}{2}\sqrt{3}+1}$	$-\sqrt{3}+1$	$\mathbf{-\sqrt{3}+\frac{1}{2}}$	$\mathbf{-\frac{1}{2}\sqrt{3}+1}$
9	$(-1,0)$	$\mathbf{-1}$	$\mathbf{-1}$	-2	-2	$\mathbf{-1}$

Table 1: *Support functions for nine unit vectors with the vertices of B.*

What can we say about the values in the table? For every unit vector u_i there is exactly one point $v \in B$ for which $\langle v, u_i \rangle$ equals the supremum, except for the unit vectors that are perpendicular to an edge of B. For instance, for every u_i in the first quadrant, that is $u_i = (x, y)$ with $x > 0$ and $y > 0$, $h(B, u_i) = \langle u_i, v_3 \rangle$. That implies that every unit vector in the first quadrant has the same support set, namely $\{v_3\}$. This statement is supported by Figure 4. No matter which unit vector in the first quadrant we choose, l will always bounce against vertex v_3.

For the unit vectors u_1, u_5, and u_9 we see that there are multiple points on B that have the same inner product with u_i. In fact, there are infinitely many. The fact that $\langle v_3, u_1 \rangle = \langle v_4, u_1 \rangle = h(B, u_1)$ implies that for every point v_j on edge e_3 $\langle v_j, u_1 \rangle$ is equal to $h(B, u_1)$. Therefore $F(B, u_1)$ is the set of all points on e_3. Figure 4e shows the case for u_5.

We can conclude that the support set for a unit vector that is the outward normal vector of an edge e_k of B is a set of points consisting of all points on e_k. The support set for any unit vector non-perpendicular to any edge of B consist of one point only, one of the vertices of B.

In Figure 4f we schematically display the support sets for B. The circle represents all unit vectors on S^1. The intersection points of the circle with the axes represent the unit vectors that are the outward normal vectors of the edges of B. These points are labeled with the edge labels from B indicating their support set.

The circle segments between the intersection points are labeled with the vertex label of the label from B that forms the support set for all unit vectors on that particular segment.

We call the representation in Figure 4f the *slope diagram* of the polygon in Figure 4a. The points on the circle representing edge e_i in the square actually is an indication for the slope of e_i.

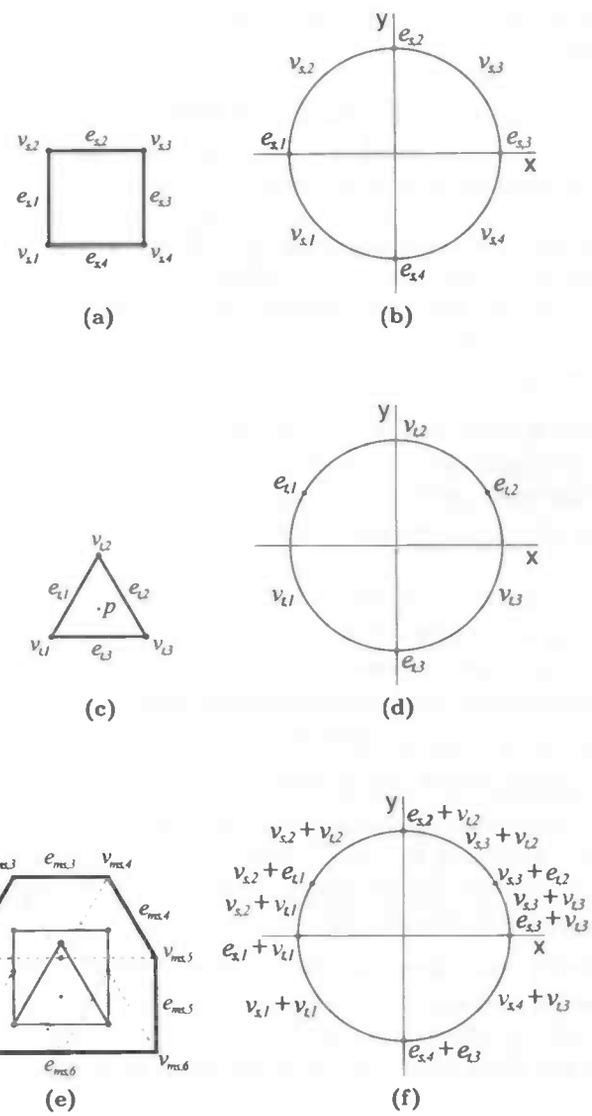


Figure 5: *The summation of the support sets of s and t*

For polyhedra in $\mathbb{R}^3$-space we can create slope diagrams as well. Clearly,

slope diagrams of 3D polyhedra are in S^2-space, that is, on the unit sphere.

Slope diagram As mentioned before we will use slope diagram representation for convex polyhedra. According to this representation faces, edges, and vertices of polyhedron P are given by points, spherical arcs, and convex spherical polygons on the unit circle, forming the slope diagram SDP.

To be more precise, SDP is a spherical graph on the unit sphere representing P as follows.

- *Face representation*: A face f_i of P which is orthogonal to the unit vector u_i is represented in SDP as the endpoint of u_i. In other words, the support set of u_i is $\{v \mid v \in f_i\}$.
- *Edge representation*: An edge e_i of P is represented in SDP by the minor arc of the great circle connecting the two points in SDP that represent the two faces adjacent to e_i. Let us call this arc $e_{SDP,i}$. In other words, the support set for every unit vector on $e_{SDP,i}$ is $\{v \mid v \in e_i\}$
- *Vertex representation* A vertex v_i of P is represented in SDP by a convex spherical polygon $f_{SDP,i}$. The polygon is bounded by the arcs in SDP that represent the edges in P that have v_i as an endpoint. In other words, the support set for every unit vector in $f_{SDP,i}$ is $\{v_i\}$.

Every convex polyhedron can be represented in the plane or on the surface of a sphere by a 3-connected *planar graph*. To be more precise, the structure of vertices and edges of a convex polyhedron can be spread out in the plane so that there are no intersections between edges.

Conversely, by a theorem of Steinitz as restated by Grünbaum, every 3-connected planar graph can be realized as a convex polyhedron [5].

Moreover, given a planar graph G, a dual graph G^* can be defined. For every node, edge, and face in G there is a face, edge, and node in G^*. In fact, this is what we just described. SDP is the dual graph of P, which can be represented as a planar graph.

By *overlaying* two slope diagrams a new slope diagram can be created. An important well known property of the Minkowski sum is that the slope diagram SDC of Minkowski sum C of polyhedra A and B is identical to the overlay of the slope diagrams SDA and SDB of A and B respectively [10], that is

$$SDC = overlay(SDA, SDB).$$

The overlay is created by superimposing the slope diagram of one of the summands over the slope diagram of the other. The positions of the points

in SDC consist of (1) the the positions of all points in SDA and SDB and (2) the positions of the intersection points of arcs of SDA and arcs of SDB. The first ones can simply be copied from the slope diagrams SDA and SDB. The latter ones are obtained during the calculation of the overlay of SDA and SDB.

The overlay SDC is the slope diagram of C, the Minkowski sum of A and B we want to calculate. Since SDC is the dual graph of C we know the structure of C. That is, we know how many vertices C contains, and which vertices are connected to each other. The vertex positions, however, are unknown. We have to calculate the positions using information from SDC.

Since slope diagrams are planar graphs, we can draw them in the plane. Figure 6a shows a fraction of SDA in the plane, and Figure 6b shows a fraction of SDB. Figure 6c shows a fraction of the overlay of SDA and SDB. In Figure 6c we see a face f_{SDC} that is the result of overlaying the faces $f_{SDA,1}$ and $f_{SDB,1}$.

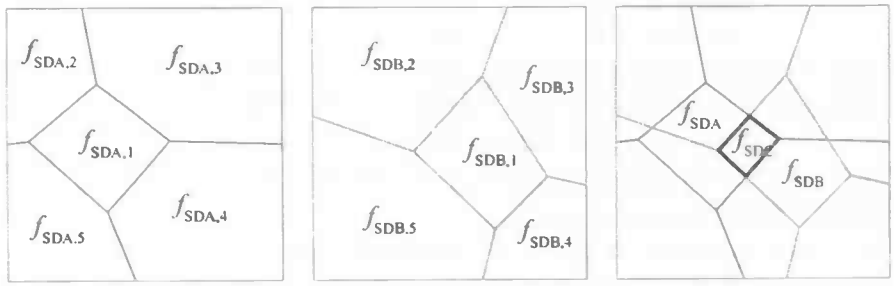


Figure 6: *Fractions of (a) slope diagram SDA, (b) slope diagram SDB, and (c) the overlay of SDA and SDB.*

Now we go back to the support function and support set. We know that face f_{SDC} in SDC represents a vertex v_C in C, but we do not know the position of v_C. We do know the positions of the vertices $v_{A,1}$ and $v_{B,1}$ which are represented in SDA and SDB by the faces $f_{SDA,1}$ and $f_{SDB,1}$, respectively. In other words, $\{v_{A,1}\}$ and $\{v_{B,1}\}$ are the support sets of $f_{SDA,1}$ and $f_{SDB,1}$. Likewise, $\{v_C\}$ is the support set for f_{SDC}.

It is known that [9]:

$$h(A \oplus B, u) = h(A, u) + h(B, u), \qquad u \in S^2,$$

and that [9]:

$$F(A \oplus B, u) = F(A, u) \oplus F(B, u), \qquad u \in S^2.$$

Hence, since $\{v_{A,1}\} \oplus \{v_{B,1}\} = \{v_{A,1} + v_{B,1}\} = \{v_C\}$ we can calculate the position of v_C by adding the vertex positions of $v_{A,1}$ and $v_{B,1}$.

In order to calculate the vertex position of every vertex $v_{C,i}$ we have to determine in which faces of SDA and SDB $f_{SDC,i}$ is located. This is called face location. Let $f_{SDC,i}$ be located in the faces $f_{SDA,j}$ and $f_{SDB,k}$ of SDA and SDB, respectively. Then vertex position $v_{C,i}$ can be calculated as follows.

$$v_{C,i} = v_{A,j} + v_{B,k},$$

where $v_{A,j}$ and $v_{B,k}$ are the vertices in A and B represented by the faces $f_{SDA,j}$ and $f_{SDB,k}$, respectively.

Using slope diagram representation is obviously the solution to the problem of redundant vertex positions. Next we will describe two methods for the calculation of the Minkowski sum of polyhedra using slope diagrams.

1.2.2 Method 2 – Using slope diagrams and planar overlay algorithms

Using slope diagram representation, as we just described it, we can avoid the calculation of irrelevant vertex positions of a Minkowski sum, that makes the naive method (section 1.2.1) inefficient. Using the overlay of two slope diagrams we only calculate vertex positions that are indeed part of the Minkowski sum.

Given two polyhedra A and B the Minkowski sum C of A and B is calculated as follows. First the slope diagrams SDA and SDB of A and B respectively are calculated. Then we use the important feature that the overlay of the slope diagrams of two polyhedra is equal to the slope diagram of the Minkowski sum of the two polyhedra. Thus calculating the overlay SDC of SDA and SDB gives us the slope diagram of C, the Minkowski sum we want to calculate.

Since there is no known efficient algorithm to calculate the overlay of two slope diagrams a switch is made from S^2 space to $\mathbb{R}^2$ space. SDA and SDB are projected on the plane, giving SDA' and SDB'. Then the planar overlay is calculated of SDA' and SDB', resulting in SDC'. This can be done in $O(n \log n)$ time [2], where n is the total number of vertices in SDA and SDB. The overlay SDC' is then projected on the sphere, having spherical overlay SDC as a result.

Several methods are used to switch from S^2 space to $\mathbb{R}^2$ space. De Raedt [4] presented a method where a slope diagram is first projected on a tetrahedron and subsequently projected from the tetrahedron on the plane. Then a planar overlay is calculated. The planar overlay is projected on a tetrahedron and finally on the sphere, giving the spherical overlay.

Fogel and Halperin [6] presented a similar method, but their method is based on projection of the slope diagrams on the unit cube. The overlays on the six separate faces of the cube are calculated. The overlays are then combined to one overlay and that overlay is then projected back on the sphere.

Once we have SDC we can construct the Minkowski sum C from SDC. With the nv vertices, ne edges and nf faces of SDC we can construct a dual graph C with nf vertices and ne edges, since SDC is a planar graph. However, the position of the nf vertices in C is unknown. Every face f_{SDC} in SDC corresponds to vertex v_c in C. But how can we calculate the position of v_c?

A face f_{SDC} in SDC is either equal to a face f_{SDA} of SDA, or it is completely contained in f_{SDA}. Likewise, f_{SDC} is either equal to a face f_{SDB} of SDB, or it is completely contained in f_{SDB}. We know the vertex positions of the vertices v_A in A and v_B in B, that correspond to f_{SDA} and f_{SDB} respectively. Another feature of the overlay of slope diagrams is that the vertex position of v_C is the sum of the vertex positions of v_A and v_B. Using face location the vertex positions of the vertices in C can be calculated by determining in which faces of SDA and SDB a face of SDC is contained.

Figure 7 shows an example of two polyhedra A and B ((a) and (b)), their slope diagram representations SDA and SDB ((c) and (d)), the overlay of SDA and SDB (e) and the Minkowski sum of A and B (f).

Face location is not very efficient and requires complex and intensive bookkeeping. Therefore we wish for a simpler method to construct the Minkowski sum from its slope diagram.

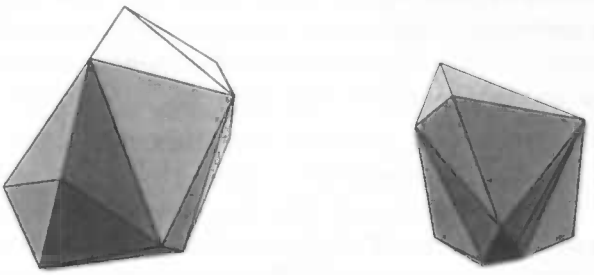
1.2.3 Method 3 – Using attributed slope diagrams and a spherical overlay algorithm

Bekker and Roerdink [1] presented a new method to efficiently calculate the Minkowski sum of two convex 3D polyhedra. Their method is based on slope diagram representation and a spherical overlay algorithm. Moreover, they use attributed slope diagrams. That is, their slope diagrams contain additional attributes that provide enough information to construct the Minkowski sum from its slope diagram, without using face location.

Let A be a polyhedron and SDA be its slope diagram. The idea is to store an edge attribute with every edge e in A containing the relative position of the target vertex of e with respect to the source vertex of e.

As we know by now, for every edge e_i in A there is an edge in SDA representing e_i. When slope diagram SDA is created the edge attribute of every edge e_i in A is copied to the edge in SDA representing e_i.

More precise, given an edge e_A in A with source vertex $source(e_A)$ and target vertex $target(e_A)$, we store an edge attribute $attr(e_A)$ with e_A, that contains



(a) (b)

Figure 7: *Polyhedra A and B.*

(c) (d)

Figure 7: *SDA and SDB, the slope diagram representations of A and B.*

(e)

Figure 7: *SDC, the overlay of SDA and SDB.*

(f)

Figure 7: *C, the Minkowski sum of A and B, can be calculated from SDC.*

the relative position of $target(e_A)$ with respect to $source(e_A)$.

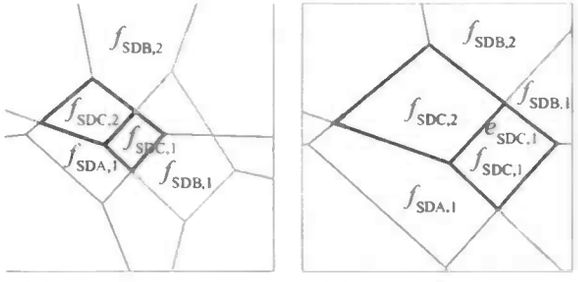
$$attr(e_A) = position(target(e_A)) - position(source(e_A))$$

During the creation of slope diagram SDA of A we store the edge attribute of every edge $e_{A,i}$ in A with edge $e_{SDA,i}$ in SDA that represents $e_{A,i}$.

$$attr(e_{SDA,i}) = attr(e_{A,i})$$

Let us take a look at the situation in Figure 8. Figure 8a shows Figure 6c again, whereas Figure 8b shows the part of Figure 8a that we are particularly interested in, we zoomed in on the faces $f_{SDC,1}$ and $f_{SDC,2}$ of SDC.

Both $f_{SDC,1}$ and $f_{SDC,2}$ are located in the same face of SDA, that is $f_{SDA,1}$. Furthermore, $f_{SDC,1}$ is located in face $f_{SDB,1}$ of SDB, whereas $f_{SDC,2}$ is located in face $f_{SDB,2}$ of SDB.



(a) *Fraction of slope diagram SDC.* **(b)** *Closer look at Figure (a).*

Figure 8: *Fractions of slope diagram SDC.*

As we have explained before, we can calculate the vertex positions of the vertices $v_{C,1}$ and $v_{C,2}$ of C, represented by $f_{SDC,1}$ and $f_{SDC,2}$, by adding the vertex positions of the vertices in A and B, represented by the faces in SDA and SDB that $f_{SDC,1}$ and $f_{SDC,2}$ are located in. Hence, the vertex positions $v_{C,1}$ and $v_{C,2}$ can be calculated as follows.

$$v_{C,1} = v_{A,1} + v_{B,1}, \tag{1}$$
$$v_{C,2} = v_{A,1} + v_{B,2}. \tag{2}$$

Clearly, when two faces $f_{SDC,i}$ and $f_{SDC,j}$ in SDC are located in the same face of SDA, the difference between the vertex positions of $v_{C,i}$ and $v_{C,j}$ is merely determined by the difference between the vertex positions of the vertices in B that are represented by the two faces in SDB that $f_{SDC,i}$ and $f_{SDC,j}$ are located in.

Edge $e_{SDC,1}$ coincides with an edge of SDB, say $e_{SDB,1}$. $e_{SDB,1}$ is adjacent to the faces $f_{SDB,1}$ and $f_{SDB,2}$, and thus represents the edge in B connecting

the vertices $v_{B,1}$ and $v_{B,2}$. Therefore the edge attribute of $e_{SDB,1}$ is set to $v_{B,2} - v_{B,1}$. The edge attribute of $e_{SDC,1}$ would be calculated as $v_{C,2} - v_{C,1}$, if the positions of $v_{C,2}$ and $v_{C,1}$ were known.

Using equations 1 and 2 we can write

$$v_{C,2} - v_{C,1} = (v_{A,1} + v_{B,2}) - (v_{A,1} + v_{B,1}) = v_{B,2} - v_{B,1}.$$

We can conclude that when an edge e_{SDC} solely coincides with an edge e_{SDA} of SDA we can copy the edge attribute from e_{SDA} to e_{SDC}.

Now we know how the edge attribute of an edge in the overlay SDC can be transferred from the original slope diagrams SDA and SDB in case the edge in SDC is solely coincides with one edge of either SDA or SDB. However, it is possible that an overlay contains edges that coincide with edges of both original slope diagrams. In Figure 9 we see fractions of SDA and SDB, and their overlay SDC, where an edge of SDC coincides with edges of both SDA and SDB.

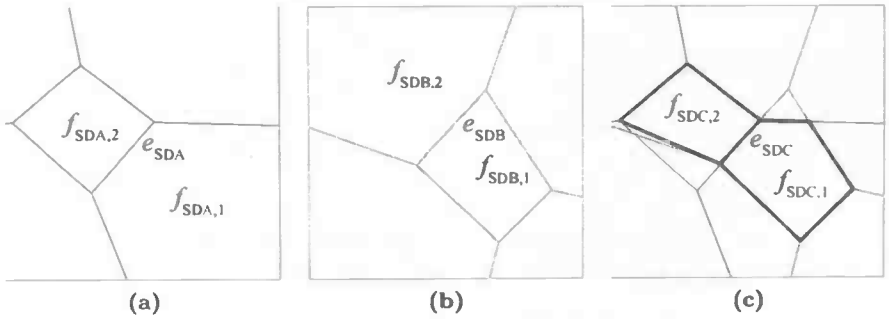


Figure 9: *Fractions of slope diagrams SDA, SDB, and SDC.*

$f_{SDC,1}$ is located in face $f_{SDA,1}$ of SDA, whereas $f_{SDC,2}$ is located in face $f_{SDA,2}$ of SDA. Furthermore, $f_{SDC,1}$ is located in face $f_{SDB,1}$ of SDB, whereas $f_{SDC,2}$ is located in face $f_{SDB,2}$ of SDB. Vertex positions $v_{C,1}$ and $v_{C,2}$ can be calculated as follows.

$$v_{C,1} = v_{A,1} + v_{B,1},$$

$$v_{C,2} = v_{A,2} + v_{B,2}.$$

$f_{SDC,1}$ and $f_{SDC,2}$ are adjacent to edge e_{SDC}, which coincides with both e_{SDA} and e_{SDB}. The edge attributes of e_{SDA} and e_{SDB} are set to $v_{A,2} - v_{A,1}$ and $v_{B,2} - v_{B,1}$, respectively. The edge attribute of e_{SDC} would be $v_{C,2} - v_{C,1}$. Using the equations from the previous paragraph, this could be written as

$$v_{C,2} - v_{C,1} = (v_{A,2} + v_{B,2}) - (v_{A,1} + v_{B,1}) = (v_{A,2} - v_{A,1}) + (v_{B,2} - v_{B,1}).$$

This means that the edge attribute of e_{SDC} can be calculated as the sum of the edge attributes of e_{SDA} and e_{SDB}, the edges of SDA and SDB that e_{SDC} coincides with.

Let us recapitulate. When the overlay SDC of two slope diagrams SDA and SDB is calculated each edge in SDC also gets an edge attribute stored with it. An edge is either (a part of) an edge of only one of the slope diagrams SDA and SDB, or (a part of) edges of both slope diagrams.

If edge e_{SDC} is solely (part of) an edge e_{SDA} in slope diagram SDA, then $attr(e_{SDC})$ is set to $attr(e_{SDA})$. If edge e_{SDC} is solely (part of) an edge e_{SDB} in slope diagram SDB, then $attr(e_{SDC})$ is set to $attr(e_{SDB})$. If edge e_{SDC} is both (part of) an edge e_{SDA} in slope diagram SDA, and (part of) an edge e_{SDB} in slope diagram SDB, then $attr(e_{SDC})$ is set to $attr(e_{SDA}) + attr(e_{SDB})$.

After SDC is calculated we can construct polyhedron C. The first part works analogue to the construction of C as we described it for method 2. The calculation of the vertex positions of C, however, is done in a different way. We do not need to locate the faces of SDC in SDA and SDB anymore.

We can transfer the edge attributes stored with every edge of SDC to the edges of C. For every edge $e_{SDC,i}$ in SDC representing edge $e_{C,i}$ in C $attr(e_{C,i})$ is set to $attr(e_{SDC,i})$.

After all edge attributes are set we can calculate the vertex positions of C. This is done as follows. Pick a random vertex $v_{C,0}$ in C and set its vertex position $position(v_{C,0})$ to a randomly chosen position, for example $(0, 0, 0)$. For each outgoing edge e_{out} from $v_{C,0}$ the vertex position of the target vertex $v_{C,t}$ of e_{out}, $position(v_{C,t})$, is set to $position(v_{C,0}) + attr(e_{out})$. Then for every outgoing edge $e_{t,out}$ of vertex $v_{C,t}$ we can set the position of the target vertex to $position(v_{C,t}) + attr(e_{t,out})$. This process is recursively repeated until every vertex position of C is calculated.

When all vertex positions of C are calculated the Minkowski sum of A and B is the result. That is, we have calculated the correct shape. The position, however, has yet to be determined. Since we only stored relative vertex positions with the slope diagrams, absolute vertex positions were lost in the process.

It can easily be checked that

$$C\_max\_x = A\_max\_x + B\_max\_x$$

should hold, where A_max_x, B_max_x, and C_max_x are the x-coordinates of the most extreme points of A, B, and C, respectively, in the positive x-direction.

C can be shifted to the correct position by three similar operations [1], one for the x-direction, one for the y-direction, and one for the z-direction.

We explain the shift-operation in the x-direction. The other operations are analogous to this one.

Before the shifting operations are applied C is at a provisional position. Let $prov\_C\_max\_x$ be the maximal x-coordinate of all nodes in C at this provisional position. Then by shifting C over

$$A\_max\_x + B\_max\_x - prov\_C\_max\_x$$

C becomes its right position in the x-direction. By applying similar shifts in the y- and z-direction C is shifted to its correct position.

The result is the Minkowski sum C of the convex polyhedra A and B.

2 Plane sweep algorithm

As mentioned in the preliminaries several algorithms are known for calculation of the Minkowski sum of convex 3D polyhedra. Some of them use a planar overlay algorithm such as the *plane sweep algorithm*, an algorithm that calculates the overlay of two sets of line segments in an efficient way. In this section we will discuss this algorithm as described in *Computational Geometry* [2]. This will make it easier to understand our 3D algorithm, that we will describe in the next section, which is based on the plane sweep algorithm.

2.1 Theory

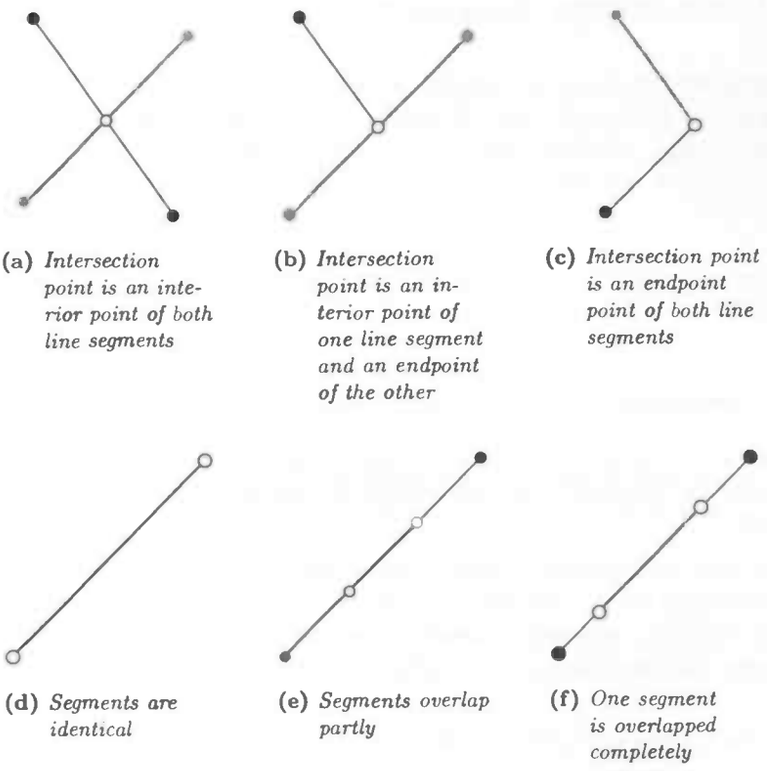
We define our problem as follows. Given two sets $S1$ and $S2$ of line segments, calculate all intersection points between a segment from $S1$ and a segment from $S2$.

Before we start we have to define what a line segments is, and furthermore what we mean by an intersection point. A line segment is closed interval on a line. Two line segments intersect when they have one or more points in common. We can distinguish five different types of intersection:

- the intersection point of two line segments is an interior point – that is, a point that is part of the segments, but not one of their endpoints – of both line segments (Figure 10a);
- the intersection point is an interior point of one of the line segments, while it is an endpoint of the other line segment (Figure 10b);
- the intersection point is an endpoint of both line segments (Figure 10c);
- the line segments are identical (Figure 10d);
- the line segments are not identical and overlap – either partly (Figures 10e) or completely (Figures 10f). That is, the line segments are parallel and at least one endpoint of one line segment coincides with an interior point of the other.

In the last two cases (Figures 10d, 10e, and 10f), in fact, there are infinitely many intersection points. However, we will only report two: either the point(s) where two endpoints coincide, or the point(s) where the endpoint of one segment coincides with an interior point of the other.

To make life easier we will combine the sets $S1$ and $S2$ by putting all their segments in one set S. We will have to redefine the problem as follows: given



(a) *Intersection point is an interior point of both line segments*

(b) *Intersection point is an interior point of one line segment and an endpoint of the other*

(c) *Intersection point is an endpoint point of both line segments*

(d) *Segments are identical*

(e) *Segments overlap partly*

(f) *One segment is overlapped completely*

Figure 10: *Six types of intersection points of two line segments. Closed dots represent regular endpoints, open dots represent intersection points*

a set S of n line segments in the plane report all intersections between the segments in S.

The easiest way to find all intersections is to compare every line segment in S with all the other segments in S. This would cost $O(n^2)$ calculations. This is very time consuming for large n.

In practice, most of the segments will intersect none or only few of the other segments. Therefore we wish for an algorithm that has a time complexity that is dependent not only on the size of the input set, but also on the size of the output set. Such an algorithm is called an *output-sensitive algorithm.* In other words, we wish for an algorithm that is more efficient than $O(n^2)$.

To make the search for intersections more efficient we have to avoid testing pairs of segments that can not intersect. Therefore we take a look at the geometry of the problem. Segments can only intersect when they are close to each other. As long as segments are far apart they can impossibly intersect.

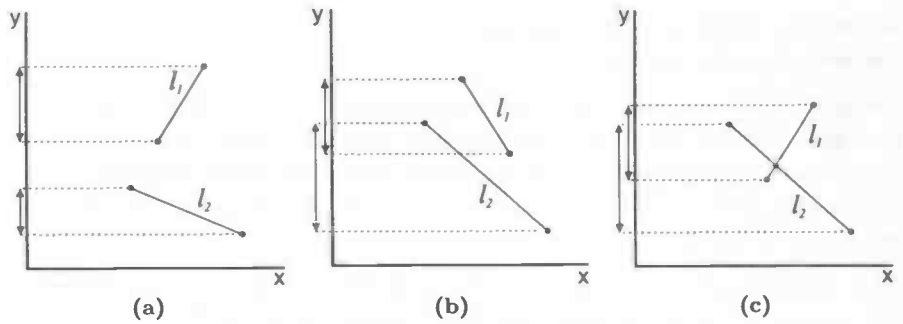


Figure 11: *(a) Non-overlapping y-intervals; (b) and (c) overlapping y-intervals*

So we want to avoid testing pairs of segments that are far apart. An easy way to this is by comparing the y-intervals. If we project the segments to compare on the y-axis we can see whether their y-intervals overlap. If this is not the case, we say that the segments are 'far' apart and do not intersect. In Figure 11a we see two line segments with non-overlapping y-intervals.

In Figure 11b we see two segments with overlapping y-intervals, but the segments do not intersect. In Figure 11c the two segments also have overlapping y-intervals, and they do intersect.

In other words, we can say that two segments can only intersect if a horizontal line exists that intersects both segments. Knowing this, we take a horizontal line above all segments. We start sweeping this line downward. During the sweep we keep track of the segments that intersect the sweep line at any time. We will discuss the details about how this works later.

For obvious reasons this type of algorithm is called a *plane sweep algorithm.* The line that is swept downward is called the *sweep line* and we will call the set of segments intersecting the sweep line at a certain moment the *status* of the sweep line.

The status of the sweep line changes during the sweep. But this does not happen continuously, it only happens when the sweep line reaches an endpoint of a segment. There are two kinds of endpoints: an endpoint where a segment starts intersecting the sweep line – let us call this kind an *upper endpoint* – and an endpoint where a segment stops intersecting the sweep line – let us call this kind a *lower endpoint.*

When an endpoint is reached the status changes. Depending on the kind of endpoint an *event* takes place. Therefore we call the upper and lower endpoints *event points.*

When an upper endpoint of a segment is reached, the segment is inserted in the status and is tested against all other segments that were already in

the status. When a lower endpoint is reached the segment it belongs to is removed from the status.

We ignored the special case that the endpoints of a horizontal line are on the sweep line at the same time. In that case we slightly slant the sweep line (Figure 13). The point that is reached first is the upper endpoint, while the other endpoint is the lower endpoint. Later we will give a more detailed definition of the ordering of points.

This method reduces the amount of calculations in general, but there are still situations thinkable where the time complexity is $O(n^2)$. For example, the situation where all segments intersect the x-axis. When we take a look at the geometry again we can find the cause of this problem in the fact that segments can still be far apart in the horizontal direction, even if they intersect the sweep line at the same time.

We can solve this problem by ordering the segments in the status in x-direction. We then only have to test new segments against the segments

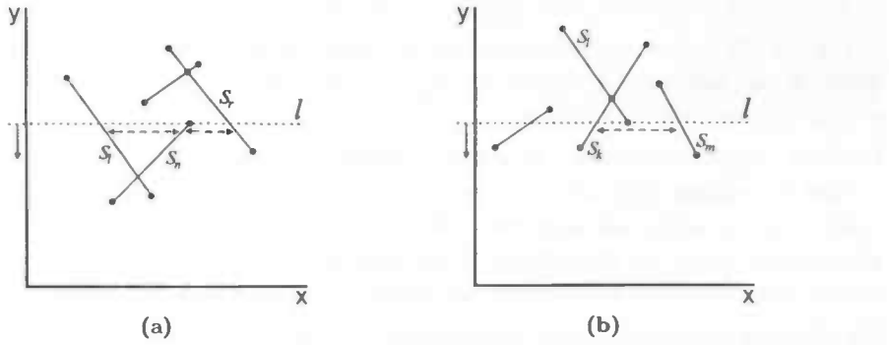


Figure 12: *The status changes (a) when a new segment is inserted after an upper endpoint is encountered, or (b) when a segment is deleted after a lower endpoint is encountered.*

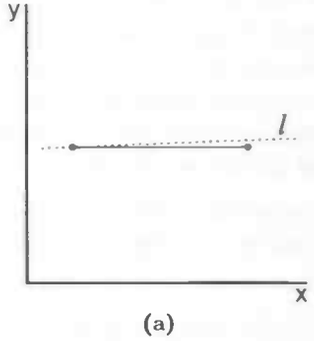


Figure 13: *Slanted sweep line.*

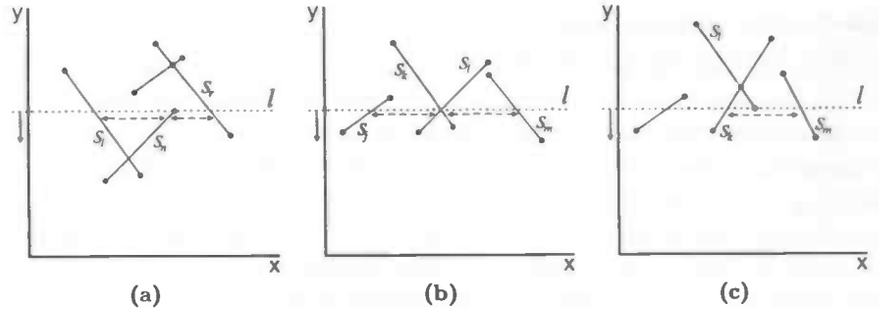


Figure 14: *New neighbors (a) after insertion of a new segment, (b) after switch segments at an intersection point, and (c) after deletion of a segment.*

that are adjacent in the status, that is, their left and right neighbor. We have to redefine the status: the status is the *ordered* sequence of segments intersecting the sweep line.

Again, the status is to be updated when the order of the segments in the status changes. As we saw in the previous paragraphs this happens when segments are either inserted in or deleted from the status. But when the sweep line reaches an intersection point of two segments the order changes as well. Therefore we introduce the intersection point as a new kind of event point.

The plane sweep algorithm now works as follows. The sweep line starts above all line segments and sweeps downward. During the sweep the algorithm takes the appropriate action when an event point is encountered. The event points are the endpoints of the segments that are known at the start of the sweep and the intersection points that are computed on the fly.

As mentioned, the actions taken by the algorithm depend on the kind of event point it detected. If an upper endpoint is found a new segment is to be inserted in the status. The new segment is tested for intersection against the segment that is directly to the left of its upper endpoint, if there is one. Then it is tested for intersection against the segment that is directly to the right of its upper endpoint, if there is one.

In Figure 14a segments s_l and s_r are adjacent segments in the status. Now segment s_n is inserted in the status, with its upper endpoint between s_l and s_r. s_n is then tested for intersections against its left neighbor s_l and its right neighbor s_r. If any intersection points are found they form new event points that will be handled later when the sweep line reaches them.

If an intersection point is found by the sweep line then the order of the segments that intersect in that point changes in the status. After the order is updated the segments that are switched have at most one new neighbor

against which they are tested for intersections.

To illustrate this Figure 14b shows the segments s_j, s_k, s_l and s_m intersecting the sweep line in this particular horizontal order. Now the sweep line encountered the intersection point of s_k and s_l and switches the order of these two segments in the status. s_j now is the new left neighbor of s_l, while s_m is the new right neighbor of s_k. The switched segments are tested against their new neighbors for intersections. These intersection points can be above or beneath the sweep line. Only intersection points beneath the sweep line are important here, while the intersection points above the sweep line have already been detected in an earlier stage of the sweep and have already been handled.

When a lower endpoint of a segment is reached the segment is removed from the status. Let s_k, s_l, and s_m be three adjacent line segments (Figure 14c). s_l will be removed when its lower endpoint is reached by the sweep line. Its left and right neighbors s_k and s_m now become adjacent and consequently have to be tested for intersection. Again, only intersection points beneath the sweep line are important.

After the sweep is completed – that is, after the last event point has been handled – all intersection points have been detected.

The invariant that stays true during the sweep is: all intersection points above the sweep line have been detected.

2.2 Data structures

Before we can discuss this algorithm in more detail we have to describe some data structures that are needed.

First of all, we need a structure to store and order the event points. We will call this the *event queue* and denote it as Q from now on. An operation is needed to fetch the next event from Q and remove it from Q. The first event point is the highest event point beneath the sweep line, that is the event point with the largest y-coordinate. If two points have equal y-coordinates the one with the smallest x-coordinate is the first event point (remember the slanted sweep line example). Another operation that we need for Q is the insertion of event points, while new event points are computed on the fly. Since it is possible that two event points coincide we also need an operation to check whether a certain event point is already present in Q.

It is reasonable to implement Q using a balanced binary search tree with ordering $\prec$, which is defined as follows. Let p and q be two event points, then $p \prec q$ – that is, p *precedes* q – if and only if $p_y > q_y$ holds or $p_y = q_y \wedge p_x < q_x$ holds.

With every event point p a set U is stored, containing the segments that have p as upper endpoint.

Using a balanced binary search tree implies that inserting, searching and deleting event points costs $O(\log m)$ time, where m is the number of event points in $\mathcal{Q}$.

We also need a data structure for the status to keep track of the segments in the status. We will call the status $\mathcal{T}$ from now on. Main purpose of this data structure is to determine the neighbors of segments to check for intersection points. This structure needs to be dynamic, since segments start and stop intersecting the sweep line. $\mathcal{T}$ contains an ordered sequence of segments so, again, a balanced binary search tree seems a good choice for data structure.

More precise, we store the segments in the leaves of the tree. An internal node of the tree contains the segment from the right most leaf of its left subtree, and is only used to determine the search path from the root of the tree to the leaf containing the requested segment. Suppose we are looking for the segment that is directly left of a point p. Starting from the root of $\mathcal{T}$ we descend $\mathcal{T}$ going left or right in a node depending whether the segment in the node is left or right from p. Eventually, we will end up in a leaf. Either the segment in this leaf, or the one in the leaf left from the current leaf contains the segment we are looking for.

The balanced binary search tree used for the status was implemented using a red-black tree [3].

2.3 The algorithm

The event queue $\mathcal{Q}$ and the status tree $\mathcal{T}$ are the only data structures we need. Now we can write down the global algorithm, as it is given in [2].

Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The set of intersection points among the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue $\mathcal{Q}$. Next, insert the segment endpoints into $\mathcal{Q}$; when an upper endpoint is inserted, the corresponding segment should be stored with it.
2. Initialize an empty status structure $\mathcal{T}$.
3. **while** $\mathcal{Q}$ is not empty
4. **do** Determine the next event point p in $\mathcal{Q}$ and delete it.
5. HANDLEEVENTPOINT(p)

The procedure HANDLEEVENTPOINT is called for every event point in $\mathcal{Q}$. We will next give the procedure, which describes how to handle event points correctly. It also clarifies what happens in cases where more than two segments intersect in one event point.

HANDLEEVENTPOINT(p)

1. Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)
2. Search in $\mathcal{T}$ for the set $S(p)$ of all segments that contain p; they are adjacent in $\mathcal{T}$. Let $L(p) \subset S(p)$ be the set of segments whose lower endpoint is p, and let $C(p) \subset S(p)$ be the set of segments that contain p in their interior.
3. **if** $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4. **then** Report p as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
5. Delete the segments in $L(p) \cup C(p)$ from $\mathcal{T}$.
6. Insert the segments in $U(p) \cup C(p)$ into $\mathcal{T}$. The order of the segments in $\mathcal{T}$ should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.
7. (* Deleting and re-inserting the segments of $C(p)$ reverses their order *)
8. **if** $U(p) \cup C(p) = \emptyset$
9. **then** Let s_l and s_r be the left and right neighbors of p in $\mathcal{T}$.
10. FINDNEWEVENT(s_l, s_r, p)
11. **else** Let s' be the leftmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
12. Let s_l be the left neighbor of s' in $\mathcal{T}$.
13. FINDNEWEVENT(s_l, s', p)
14. Let s'' be the rightmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
15. Let s_r be the right neighbor of s'' in $\mathcal{T}$.
16. FINDNEWEVENT(s'', s_r, p)

FINDNEWEVENTS(s_l, s_r, p)

1. **if** s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection point is not yet present as an event in $\mathcal{Q}$
2. **then** Insert the intersection point as an event in $\mathcal{Q}$.

2.4 Implementation

As an exercise we implemented the plane sweep algorithm using C++ and exact numbers. LEDA – Library of Efficient Data types and Algorithms [8] – was used. For the visualization of the algorithm openGL was used.

2.5 Results

Now we have implemented the plane sweep algorithm as described by Berg et al. [2] it is time to run some tests.

The algorithm is expected to run in $O(N \log N + I \log N)$. We also expect that the summand $I \log N$ is dominant when the number of intersections is

N	I	$t(s)$	$N \log N$	$I \log N$	c_1	c_2	c_3
40	193	0.04	213	1,027	3.23E-05	2.50E-05	2.07E-04
50	305	0.07	282	1,721	3.49E-05	2.80E-05	2.30E-04
75	629	0.14	467	3,918	3.19E-05	2.49E-05	2.23E-04
100	1,054	0.26	664	7,003	3.39E-05	2.60E-05	2.47E-04
150	2,471	0.65	1,084	17,862	3.43E-05	2.89E-05	2.63E-04
200	4,130	1.12	1,529	31,569	3.38E-05	2.80E-05	2.71E-04
300	9,757	2.95	2,468	80,288	3.56E-05	3.28E-05	3.02E-04
400	17,278	5.35	3,458	149,349	3.50E-05	3.34E-05	3.10E-04
500	28,071	8.99	4,483	251,679	3.51E-05	3.60E-05	3.20E-04
1000	113,588	41.73	9,966	1,131,994	3.65E-05	4.17E-05	3.67E-04
2000	458,504	190.48	21,932	5,027,856	3.77E-05	4.76E-05	4.15E-04

Table 2: *Results of our implementation of the plane sweep algorithm. N is the number of segments, I is the number of intersection points, t is the time the algorithm needed to compute all intersection points. The constants c_1, c_2 and c_3 are calculated as $c_1 = \frac{t}{(N+I)\log N}$, $c_2 = \frac{t}{N^2}$, and $c_3 = \frac{t}{I}$. Comparing c_3 for $N = 40$ and $N = 2000$ having the values 2.30×10^{-4} and 4.15×10^{-4} respectively makes it reasonable to conclude that c_3 is fairly constant.*

large. This is obviously the case when the segments are long. On the other hand, when the segments are short the number of intersections is small and $N \log N$ will be the dominant summand. Clearly, since we used binary search trees for the implementation, when we speak about log we actually mean ${}^2\log$.

We start by running the program with randomly generated segments and different values of N. The results are shown in Table 2.

N is the number of segments in the input set, I the number of intersection points found by the algorithm, and t is the time the algorithm needed to find all intersection points.

The fourth and fifth column contain $N \log N$ and $I \log N$, which are combined to determine the time complexity of the algorithm.

The sixth column of the table of results contains c_1, computed as $c_1 = \frac{t}{(N+I)\log N}$. Since the algorithm should run in $O((N+I)\log N)$ time, we expect c_1 to be constant for every value of N. The seventh column contains c_2, computed as $c_2 = \frac{t}{N^2}$. We would expect c_2 to decrease, since N^2 grows compared to $O((N+I)\log N)$. But we see something unexpected happen. c_2 is nearly as constant as is c_1, and even increases when N grows large.

Studying the table learns us that this is not as strange as it seems. Taking randomly chosen endpoints for the segments resulted in large numbers of

N	I	$t(s)$	$N \log N$	$I \log N$	c_1	c_2	c_3
50	80	0.01	282	452	1.36E-05	4.00E-06	1.25E-04
100	266	0.06	664	1,767	2.47E-05	6.00E-06	2.26E-04
200	1,036	0.26	1,529	7,919	2.75E-05	6.50E-06	2.51E-04
500	7,364	2.10	4,483	66,024	2.98E-05	8.40E-06	2.85E-04
1000	29,261	9.45	9,966	291,609	3.13E-05	9.45E-06	3.23E-04
2000	117,455	42.75	21,932	1,287,986	3.26E-05	1.07E-05	3.64E-04
3000	268,655	104.97	34,652	3,103,166	3.35E-05	1.17E-05	3.91E-04
4000	470,453	195.31	47,863	5,629,339	3.44E-05	1.22E-05	4.15E-04
5000	656,696	283.30	61,439	8,069,292	3.48E-05	1.13E-05	4.31E-04

Table 3: *Results of our implementation of the plane sweep algorithm with SEG_SCALE*= 0.5

long segments, which results in a large number of intersections. In fact, the number of intersections I grows quadratically with N. More precise, $I \approx \frac{N^2}{75} \log N$ for $N = 200$.

Clearly, when a segment is enlarged to be twice as long (or in other words, a segment just as long is added), it will in general intersect twice as many segments. We added a constant c_3 calculated as $c_3 = \frac{t}{I}$. Comparing c_3 for $N = 40$ and $N = 2000$ having the values 2.30×10^{-4} and 4.15×10^{-4} respectively makes it reasonable to conclude that c_3 is indeed fairly constant. After all, c_3 hardly doubles, whereas N grows a factor 50.

Since the number of intersections is related to the number of segments we can also say that it is related to the length of the segments. Therefore we can decrease the number of intersections by scaling the segments by a factor SEG_SCALE.

The results for SEG_SCALE with a value of 0.5 are in Table 3. Table 4 shows the result for a value of 0.2 for SEG_SCALE. For SEG_SCALE = 0.5 we would expect $I \approx \frac{N^2}{300} \log N$ for $N = 200$, which is indeed affirmed in Table 3. Likewise, Table 4 shows that $I \approx \frac{N^2}{1875} \log N$ holds for SEG_SCALE = 0.2.

We see that the summand $I \log N$ is indeed less dominant when the length of the segments decreases. This is interesting, because in slope diagrams the length and number of edges are directly dependent on the complexity of the polyhedra they represent. The larger the number of faces of polyhedron P, the larger the number of edges in slope diagram SDP and the shorter the edges are. This is different in the tests we did for the plane sweep algorithm. Using randomly generated line segments implied that the length of the segments and number of segments are independent, and so are the number of intersections and the length of the segments.

We can conclude the following. Firstly, the algorithm runs in $O((N +$

N	I	$t(s)$	$N \log N$	$I \log N$	c_1	c_2	c_3
100	36	0.02	664	239	2.21E-05	2.00E-06	5.56E-04
200	143	0.06	1,529	1,093	2.29E-05	1.50E-06	4.20E-04
500	937	0.38	4,483	8,401	2.95E-05	1.52E-06	4.06E-04
1000	3,688	1.64	9,966	36,754	3.51E-05	1.64E-06	4.45E-04
2000	14,670	7.25	21,932	160,868	3.97E-05	1.81E-06	4.94E-04
5000	91,548	53.05	61,439	1,124,915	4.47E-05	2.12E-06	5.79E-04

Table 4: *Results of our implementation of the plane sweep algorithm with SEG_SCALE*= 0.2

$I) \log N)$ time.

Secondly, it takes between 1×10^{-4} and 5×10^{-4} second per intersection to calculate the overlay of two sets of line segments.

3 Sphere sweep algorithm

In order to efficiently calculate the Minkowski sum of two polyhedra we need to calculate the overlay of the slope diagrams of the polyhedra, giving the slope diagram of the desired Minkowski sum.

Since the plane sweep algorithm, as we described it in the previous section, is an efficient method to calculate the planar overlay of a set of line segments, we will take it as the basis for our own algorithm.

We define our problem as follows. Given two slope diagrams S_1 and S_2 calculate the overlay S of S_1 and S_2.

Switching from line segments in two-dimensional space to spherical arcs in S^2 is nontrivial. We will have to deal with some difficulties. Firstly, we have to define how event points and spherical arcs are represented. Secondly, we have to define an ordering on event points and spherical arcs, and we have to explain how intersection points between spherical arcs are calculated. And finally, we have to choose another sweep line representation and another kind of sweep movement.

3.1 Representation of points and edges of slope diagrams

Let A be a polyhedron and SDA the slope diagram of A. Theoretically, points of a slope diagram are the endpoints of unit vectors. More precise, a point p_i of SDA is the endpoint of the unit normal vector u_i perpendicular to a polyhedral face f_i of A. However, since we will use rational numbers for exact mathematics, a unit normal vector is, in general, unavailable. This is not an impassable problem. It is the direction of the unit vector that we are interested in, after all a point in a slope diagram is an indication for the slope of a polyhedral face. Therefore, for practical purposes, we can use the regular normal vector of f_i to represent f_i in SDA. The projection of this normal vector on the unit sphere is the unit vector p_i.

An edge e in SDA is the minor arc of the great circle through two points p_1 and p_2. As mentioned p_1 and p_2 are unit normal vectors of two faces in A. But in practice we will use the regular normal vectors p'_1 and p'_2. e then is the projection of the line segment between p'_1 and p'_2 on the unit sphere. Again, in theory, when we speak about edge e we mean an arc on the unit sphere, whereas in practice we use the line segment between p'_1 and p'_2.

3.2 Sweep line

For obvious reasons we can not use a straight line as sweep line in our algorithm. We need a curved line on the sphere. But what curve is suitable for use as sweep line? We will now take some options into consideration.

Before we start our search for an appropriate sweep line we note that we call the upper most point of the sphere the *north pole*, and that we will refer to the lower most point of the sphere as the *south pole* as if we deal with a globe.

Let us take a horizontal plane that moves downward from a position above the sphere ($y > 1$) to a position underneath the sphere ($y < -1$). The intersection with the sphere is a circle. At the north pole of the sphere it is a point, that is, a circle with radius 0. From there, as the plane moves downward, the circle grows until it has a radius 1 when the plane is in the xz-plane ($y = 0$). From then on, the radius decreases until the plane reaches the south pole where the intersection circle has a radius 0 again.

Taking the intersection circle as the sweep line would cause one big problem. Imagine a circle segment passing the north pole. This segment would intersect the sweep line twice in case the sweep line was between the north pole and the highest endpoint of the segment. Therefore this method is not suitable for our algorithm.

The previous problem is caused by the intersection circle not being a great circle. Taking a great circle would solve the problem. Imagine a great circle C on the unit sphere. Now imagine two points p_1 and p_2 on the unit sphere – at most one of them on C. Assume that the origin O of the unit sphere, and p_1 and p_2 are not collinear. The minor arc of the great circle through p_1 and p_2 can not intersect C in more than one point.

Therefore it is evident to take a great circle on the unit sphere as the sweep line. Let us take the great circle through the north and south pole of the unit sphere and rotate it around the y-axis. Since the angle between two endpoints is always less than π, it is impossible that one segment intersects the sweep line twice. For the sake of simplicity, we ignore some rare cases where circle segments indeed intersect the sweep line twice or more.

Unfortunately, this approach causes another problem. Rotating the whole great circle in fact implies that we keep track of two statuses. Let us explain. Figure 15a shows the unit sphere from above. The sweep line has been rotated 35° and the segments s_1 and s_2 have their starting endpoints on the sweep line. Now we split the sweep line at the north pole and spread it out to be a straight line segment. The two endpoints of the line are in fact the same point of the actual sweep line, that is, the north pole. The midpoint is the actual south pole. Figure 15b shows the straight line representation of the sweep line and s_1 and s_2. Conspicuously, s_1 and s_2 have opposite directions.

We defined the status as the the sequence of segments that intersect the sweep line at the same time. Moreover, segments that are in the status are close. More precise, they are close enough to intersect. However, segments having their starting endpoints on the 'upper half' of the sweep line, and

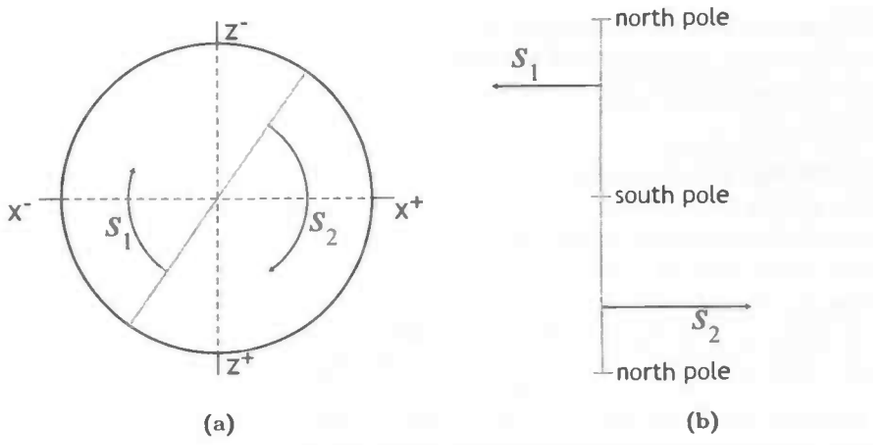


Figure 15: *The sphere viewed from above the north pole.*

segments having their starting endpoints on the 'lower half' of the sweep line obviously can impossibly intersect. Note that for now we ignore the degenerate cases where segments intersect either the north or the south pole.

Although the last approach is not suitable for our algorithm, it steers us in the right direction. That is, we can use a variant on the rotating great circle. We can sweep half of the great circle around the y-axis. Our sweep line is a semicircle segment between the north pole and the south pole of the unit sphere.

As the sweep line rotates it encounters all the points on the sphere. For now we ignore some nasty cases, like segments starting or ending in one of the poles, and segments that pass through one of the poles.

Now that we have decided what kind of sweep line we will use, we immediately encounter a new problem. Whereas in the plane we can easily find a position above all segments, this is not possible when sweeping a slope diagram. Whatever initial position we choose for the sweep line, it will always intersect one or more segments of the slope diagram, either in their endpoints or in interior points.

We can solve this problem by ignoring all segments intersecting the sweep line in its initial position in another point than their upper endpoint. Their upper endpoint will be encountered later on during the sweep. Now we will sweep the sphere twice, which in fact means we process all segments twice. We will call the second phase of the sweep the *encore* sweep. When we have rotated the sweep line 2π rad around the y-axis it is in its initial position again and the segments that we ignored earlier are now in the status. So if we do the sweep again, it will find intersection points, if any, of the segments

in the status with the segments that we already processed.

We can do this without doing harm to the time complexity, while – assuming our algorithm will run in $O(n \log n) - O(2n \log 2n)$ is still $O(n \log n)$.

When we take a closer look at this method we see that, in case a segment has an upper endpoint that lies before the initial position of the sweep line, its lower endpoint will be encountered before the sweep line has been rotated by π rad. That means that rotating the sweep line by 3π is enough to find all intersections.

The invariant from the plane sweep algorithm still holds: all intersection points above the sweep line have been detected. Of course, only intersection points between segments that are processed can be detected! That implies that during the encore sweep we will only find possible intersection points between segments of which the upper endpoint was found during the first sweep and any other segments. Therefore we can abort the encore sweep when we processed the last segment that was in the status after the first sweep was finished. How this is done will be explained in detail later.

The time complexity of our sphere sweep algorithm is $\Omega(1\frac{1}{2}n \log 1\frac{1}{2}n)$

3.3 Event points and their ordering

We are already familiar with the representation of points of slope diagrams. In our sphere sweep algorithm event points are the points of the slope diagrams and the intersection points of edges of one slope diagram with edges of the other slope diagram. Since points in our slope diagrams are in fact only directions, defining an order of event points is a little more complicated than for the plane sweep algorithm, where we could just compare x- and y-coordinates. We have to think of another method to determine the order of event points.

Recall that in the plane an order $\prec$ was defined on event points so that with two event points p and q we have $p \prec q$ if and only if $p_y > q_y$ holds or $p_y = q_y \wedge p_x < q_x$ holds. $p_y > q_y$ determines which point is encountered first by the sweep line, assuming the points do not have the same height.

We will have to find a way to translate this to our algorithm. In the sphere sweep algorithm we do not sweep the sweep line from a certain position downward. Instead the sweep line is rotated around the y-axis. So $p_y > q_y$, indicating that p will be handled before q while the sweep line encounters it first, could be translated to our algorithm by comparing the rotation angle of the points around the y-axis. To be more precise, we project the points p and q on the xz-plane and compare the angles of these projections p' and q' with the initial sweep line position. That is, we compare the angles that p' and q' make with the point $(0, 0, 1)$. (Figure 16a) The point with the

smallest angle will be encountered by the sweep line first and thus is the smaller event point. We will call the angle of p' with $(0,0,1)$ p_α.

In the plane $p_y = q_y \wedge p_x < q_x$ means that both points are on the sweep line at the same time. The x-coordinate then determines which point is to be handled first. This is easy to translate to our sphere sweep algorithm. When two event points are on the sweep line at the same time – they have equal angles with the initial sweep line position – we simply compare their y-coordinates.

But wait, did we not use regular normal vectors that are, in general, unequal to the unit normal vectors? Unfortunately this makes it impossible to simply compare the y-coordinates. The solution is quite straightforward. Instead of comparing the y-coordinates of p and q, we compare the angles of p and q with the xz-plane. We call the angle of point p with the xz-plane p_β.

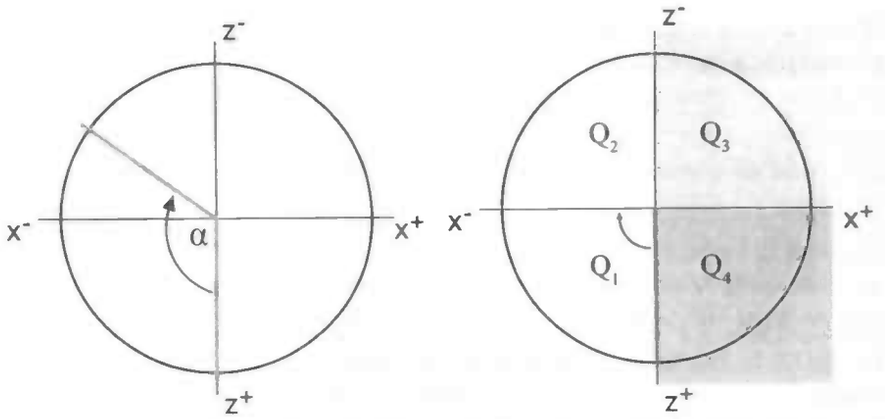
We can now define the order of two event points p and q as $p \prec q$ if and only if $p_\alpha < q_\alpha$ holds or $p_\alpha = q_\alpha \wedge p_\beta > q_\beta$ holds.

In theory we can use this order, but in practice we have to overcome some difficulties. Since we use rational numbers for exact computations some trigonometric functions – sine and cosine in particular – are unavailable and thus we can not simply solve this problem by using spherical coordinates. The tangent is available for use since it can be calculated by dividing two coordinates. But even though we can calculate the tangent, we still can not calculate the angle. This however is not a problem. We can just store the tangent of the angle instead the angle itself. So p_α contains the tangent of the angle of p' with $(0,0,1)$. Likewise, p_β contains the tangent of the angle of p with the xz-plane.

Figure 16a shows the sphere viewed from above. We can calculate the tangent by dividing the z-coordinate by the x-coordinate. In combination with the sign of the x-coordinate we can determine the rotation angle of p' with the xy-plane. If $-1 \leq p_x < 0$ holds the sweep line will encounter p in the first half of the sweep, if $0 < p_x \leq 1$ holds in the second half. We still have a problem: the tangent is undefined when $p_x = 0$, that is, for angles of $\frac{1}{2}\pi$ rad and $-\frac{1}{2}\pi$ rad.

We can avoid this problem by splitting the sphere in four quarters (Fig. 16b). Let us call the quarters Q_1, Q_2, Q_3, and Q_4. In its initial position the sweep line intersects the xz-plane in point $p_{init} = (0,0,1)$. l_α is the rotation angle of the sweep line. The sweep line is in the first quarter if $0 \leq l_\alpha < \frac{1}{2}\pi$ holds, in the second quarter if $\frac{1}{2}\pi \leq l_\alpha < \pi$ holds, in the third quarter if $\pi \leq l_\alpha < 1\frac{1}{2}\pi$ holds, and in the fourth quarter if $1\frac{1}{2}\pi \leq l_\alpha < 2\pi$ holds.

For every event point we do not store the tangent of the angle of its projection on the xz-plane with p_{init}, but only the tangent of the angle within its own quarter. That is, for a point p in Q_1 p_α is the tangent of the angle p' with the positive z-axis, for a point p in Q_2 p_α is the tangent of the angle



(a) *Angle α with the initial sweep line position.*

(b) *In order to avoid division by zero we have to split the xz-plane in four quarters.*

Figure 16: *The rotation angle of a point in the slope diagram with the initial sweep line position.*

of p' with the negative x-axis, etc. Therefore $0 \leq \arctan p_\alpha < \frac{1}{2}\pi$ holds for any event point.

We can now redefine the order $\prec$ as follows. With p_Q being the index of the quarter containing point p, $p \prec q$ if and only if $p_Q < p_Q$ holds, or $p_Q = p_Q \wedge p_\alpha < q_\alpha$ holds, or $p_Q = p_Q \wedge p_\alpha = q_\alpha \wedge p_\beta > q_\beta$ holds.

In this order we have ignored the fact that event points can be handled a second time during the encore sweep. After handling an event point for the first time we want to insert the same event point at the end of the event queue. We have to change the order $\prec$ again to make a distinction between an event point in the first sweep and the same event point in the second sweep. Therefore we add another property to the event point: the *encore number*. An event point has encore number 0 as long as it has not been handled yet. After an event point is handled its encore number is increased to 1 and the 'new' event point is inserted in the event queue.

Taking the encore number into account we can give a final definition of the order $\prec$. With p_{enc} being the encore number of p we have $p \prec q$ if and only if $p_{enc} < p_{enc}$ holds, or $p_{enc} = p_{enc} \wedge p_Q < p_Q$ holds, or $p_{enc} = p_{enc} \wedge p_Q = p_Q \wedge p_\alpha < q_\alpha$ holds, or $p_{enc} = p_{enc} \wedge p_Q = p_Q \wedge p_\alpha = q_\alpha \wedge p_\beta > q_\beta$ holds.

3.4 Spherical arcs and intersection points

The most important difference between the plane sweep algorithm and the sphere sweep algorithm is the way to determine whether segments intersect.

Whereas it is trivial to calculate the intersection of two segments in the plane, this is not the case in three dimensional space.

In theory the intersection point of two spherical arcs can be calculated using trigonometric functions. But in practice this is not possible. Again, for our algorithm we use exact numbers, and hence trigonometric functions are not available.

We have to find a method to compute the intersection point of spherical arcs without using trigonometric functions. Therefore we have to take a closer look at what the points and arcs of a slope diagram are.

Let SDA be the slope diagram of polyhedron A. Let s_1 be an arc of slope diagram SDA connecting two noncollinear points p_1 and p_2. p_1 is the unit normal vector of a face f_1 in A. In general, the exact position of p_1 can not be determined, since that would require trigonometric functions, which are unavailable. What we do know is the regular normal vector v_1. In fact, p_1 is the intersection point of the straight line through O and v_1 with the unit sphere. Likewise, p_2 is the intersection point of a line through O and v_2 with the unit sphere, where v_2 is the normal vector of a face f_2 of A adjacent to f_1.

Plane P_1 spanned by the half-lines $\vec{Ov_1}$ and $\vec{Ov_2}$ intersects the unit sphere in a minor arc of the great circle through p_1 and p_2. In fact, s_1 is a projection of l_1, the line segment between v_1 and v_2 on the unit sphere.

Analogously, let SDB be the slope diagram of polyhedron B. Let s_2 be an arc of SDB, non-parallel to s_1, with endpoints p_3 and p_4. s_2 is the projection on the unit sphere of l_2, the line segment between the vectors v_3 and v_4, the normal vectors of two adjacent faces f_3 and f_4 in B. P_2 is the plane spanned by the half-lines $\vec{Ov_3}$ and $\vec{Ov_4}$

Let p_i be the intersection point of s_1 and s_2. The half-line $\vec{Op_i}$ intersects both s_1 and s_2 in p_i. Since s_1 and s_2 are projections on the unit sphere of l_1 and l_2, $\vec{Op_i}$ intersects these line segments as well. We can use this information to create a system of equations to determine an equation for the line l_i through O and p_i.

Two arbitrary points p_j and p_k on the line segments l_1 and l_2, respectively, can be written as follows.

$$p_j = \lambda_1 p_1 + (1 - \lambda_1) p_2, \qquad 0 \leq \lambda_1 \leq 1, \tag{3}$$

$$p_k = \lambda_2 p_3 + (1 - \lambda_2) p_4, \qquad 0 \leq \lambda_2 \leq 1. \tag{4}$$

The constraint $0 \leq \lambda_1 \leq 1$ indicates that for p_j only points between p_1 and p_2 are taken into account. Likewise for λ_2 and p_j.

If s_1 and s_2 intersect there exists a half-line l_i bound by the origin O intersecting s_1 and s_2 and thus intersecting l_1 and l_2. Then for certain values of

λ_1 and λ_2 p_j and p_k are the intersection points of l_i with l_1 and l_2 respectively. In fact, we could say that p_j and p_k are the same vector, possibly with different lengths. We can write $p_j = \lambda_3 p_k$. Using equations 3 and 4 we can write

$$\lambda_1 p_1 + (1 - \lambda_1)p_2 = \lambda_3(\lambda_2 p_3 + (1 - \lambda_2)p_4). \tag{5}$$

Equation 5 is in fact a set of 3 linear equations in 3 unknowns, λ_1, λ_2, and λ_3. Since the vectors p_1, p_2, p_3, and p_4 are linearly independent, the system has a unique solution.

Using equation 5 we will find the values for λ_1 and λ_2 for which p_j and p_k are collinear with O. However, it is still possible that s_1 and s_2 do not intersect. In case $\lambda_3 < 0$ O, p_j and p_k are collinear, but p_j and p_k point in opposite directions. Therefore only when $\lambda_3 > 0$ holds p_j and p_k have the same direction and thus an intersection point of s_1 and s_2 is detected.

3.5 Degenerate cases

So far we ignored some nasty cases, which make the sweep algorithm more complicated. First of all it is possible that a segment has the north pole as its upper endpoint or the south pole as its lower endpoint. The main problem in these cases is the fact the north and south pole can not be ordered using the order $\prec$ as we defined it before. The x- and z-coordinates of these points are both equal to zero and therefore p_α and p_β can not be calculated, since it would imply a division by zero.

Another difficulty is the fact that the upper endpoint of a segment starting at the north pole always lies on the sweep line and hence the segment always intersects the sweep line, even after the lower endpoint has been handled.

A similar problem occurs when a segment has the south pole as its lower endpoint. The segment always intersects the sweep line, even if the upper endpoint has not been handled yet.

These problems can easily be solved by using the following conventions.

- The north pole is the first event point during a single sweep if at least one segment has it as its upper endpoint. The property p_Q is set to 0, so that the event point will be handled before any other event point having the same encore number. The properties p_α and p_β are irrelevant and set to 0 to avoid division by zero during the initialization of the event point.
- The south pole is the last event point during a single sweep if at least one segment has it as its lower endpoint. The property p_Q is set to 5, so that the event point will be handled after any other event point having the same encore number. The properties p_α and p_β are irrelevant and

set to 0 to avoid division by zero during the initialization of the event point.

A bigger problem occurs when a segment passes through the north or south pole. That is, when it has either the north or the south pole as an interior point. Let s be the segment passing through the north pole with $p_1 = (-\frac{1}{3}\sqrt{3}, \frac{1}{3}\sqrt{3}, \frac{1}{3}\sqrt{3})$ and $p_2 = (\frac{1}{3}\sqrt{3}, \frac{1}{3}\sqrt{3}, -\frac{1}{3}\sqrt{3})$ as its endpoints. Clearly, p_1 is the upper endpoint, since $p_{1,Q} = 1$ and $p_{2,Q} = 3$. But ordering the endpoints like this makes s invalid. The points on s between the north pole and p_1 lie 'above' the sweep line. That means that those points are encountered by the sweep line before it reaches the upper endpoint. This goes against the basics of the algorithm.

Reversing the order of the endpoints in these particular cases would not solve the problem, since it would only postpone the problem to occur when the sweep line encounters p_2.

The only solution to this problem is splitting s at the north pole. Two new segments are created, both with the north pole as their upper endpoint and with p_1 and p_2 as their respective lower endpoints.

Consequently, the north pole is detected as an intersection point. If the north pole is indeed an intersection point, while at least one other segment – non-parallel to s – either has the north pole as an endpoint, or passes through the north pole, the algorithm would run correctly. However, if s is the only segment passing through the north pole, or a segment from the other slope diagram has the north pole as an interior point, the north pole would incorrectly be reported as an intersection point. It can be easily checked if this situation occurs when the north pole is handled.

The situation, where only one segment or two parallel segments have the south pole as an interior point is very similar to the one that was just described. These segments are split in the south pole, creating two new segments with the south pole as their lower endpoints and the other endpoints as their respective upper endpoints.

3.6 Data structures

For the implementation of our sweep algorithm we can use the exact same data structures as for the plane sweep algorithm. That is, we use a status tree that stores circular segments instead of line segments and an event queue that stores event points as we defined them in section 3.3. Since using these data structures made the plane sweep algorithm run in $O(n \log n)$, we expect our algorithm to have this time complexity as well.

What we do have to change are the various comparison functions for event points and circle segments.

3.7 The algorithm

The sphere sweep algorithm works analogously to the plane sweep algorithm. We can refer section 2.3 for details about the algorithm.

3.8 Constructing the overlay

The result of our sphere sweep algorithm is a set S of event points representing the points in slope diagram SDC of the Minkowski sum C of A and B. Furthermore, we have a list L of all edges in slope diagrams SDA and SDB. With every event point p in S we have stored the sets p_U, p_L, and p_C of references to the edges in L. p_U contains references to the edges in L that have point p as their upper endpoint. Likewise, p_L contains references to the edges in L that have point p as their lower endpoint, and p_C contains references to the edges in L that have point p as an interior point.

With S, L and the sets p_U, p_L, and p_C of every point in S we have enough information to construct the overlay SDC of SDA and SDB.

We have the points of SDC already, they are all the points in S. All we have to do to create the correct overlay is determine which points in S are to be connected.

We know, that every edge in SDC is either (a part of) an edge of SDA, (a part of) an edge of SDB, or (a part of) an edge of both SDA and SDB. That means that we can use the information about the edges referenced in p_U, p_L, and p_C to determine which connections we need to add to SDC.

More precise, when a certain edge $e_{SDA,i}$ is referenced in set p_U, it means that p is the upper endpoint of $e_{SDA,i}$. Moreover, $e_{SDA,i}$ connects p and a certain point q in S that is either the other endpoint of $e_{SDA,i}$, or an interior point of $e_{SDA,i}$ being the intersection point of $e_{SDA,i}$ with another edge. Obviously, in case there are more than one intersection points on $e_{SDA,i}$, q is the closest one to p. The case where $e_{SDA,i}$ is referenced in the set p_L is analogue to the previous one.

When $e_{SDA,i}$ is referenced in p_C, it means that p is an intersection point of $e_{SDA,i}$ with one or more edges. In this case, p is connected to two other points q_1 and q_2 on $e_{SDA,i}$, which of course are on opposite sides of p. Again, q_1 can be either an endpoint of $e_{SDA,i}$ or the closest intersection point of $e_{SDA,i}$ with another edge. Likewise for q_2.

The same explanation applies to edges of slope diagram SDB. It is possible that two points p and q are connected by (a part of) an edge of SDA as well as (a part of) an edge of SDB. Clearly, we only add one connection to SDC. Nevertheless, it is important to know whether this is the case. Recall, that we transfer edge attributes from SDA and SDB to SDC. When p and

q are connected by the edges $e_{SDA,i}$ and $e_{SDB,j}$ both their edge attributes are transferred to the edge connecting p and q.

For the construction of the overlay SDC of SDA and SDB we can use a sweep similar to the one used previously for the calculation of the intersection points. We order the event points in S the same way as during the sphere sweep algorithm. Also, after handling a point, the encore number of the point is increased and the point is reinserted in the set S again.

As mentioned every event point p in S contains three sets p_U, p_L, and p_C. Each edge referenced in p_U, p_C, or p_L, has a property *lastIP* that keeps track of the last intersection point on the edge that has been handled during the sweep. Initially, *lastIP* is set to the null-vector for all edges.

Event point p is handled as follows. If p is an upper endpoint, that is, p_U contains at least one reference to an edge ($|p_U| > 0$), we have to do only one thing for the edge(s) referenced in p_U. Property *lastIP* for every edge referenced in p_U is set to the current event point.

If p is an interior point ($|p_C| > 0$), or a lower endpoint ($|p_L| > 0$), for every edge e referenced in p_C and p_L e_{lastIP} is checked. If e_{lastIP} equals NULL no intersection point on e has been handled yet. This is the case if the 'previous event point' on s lies 'above' the initial sweep line position. We can ignore this edge for now, it will be handled after the upper endpoint is found later on during the sweep. p will be handled in the second sweep again. Therefore the encore number for p is increased and p is reinserted in S.

If $e_{lastIP} \neq NULL$ then the part of e starting at e_{lastIP} and ending in p is an edge of the overlay SDC, and thus it is added here. The reversal edge of the new edge is added as well.

When the appropriate action is taken for e and p is an interior point of e e_{lastIP} is set to p. When all edges referenced in $p_U \cup p_C \cup p_L$ are handled the next event point is handled.

For all event points left in S after the first sweep is finished the actions are repeated, but only for those edges referenced in $p_U \cup p_C \cup p_L$ with *lastIP* set to a point that lies 'above' the initial sweep line position.

We now have all the points and edges in SDC. Using LEDA we compute all faces of SDC.

3.9 Constructing the Minkowski sum from its slope diagram

We now have the slope diagram SDC of the Minkowski sum C from polyhedra A and B. We have all the information we need to construct C from SDC.

Let us start with a new polyhedron with no points. We already know that every face in SDC represents a node in C. Therefore we add a node to C

for every face in SDC. The positions of the nodes will be computed later using the edge attributes we have stored earlier.

We also know that if two faces $f_{SDC,i}$ and $f_{SDC,j}$ in SDC are adjacent the nodes in C represented by $f_{SDC,i}$ and $f_{SDC,j}$ are connected by an edge. For every face edge e of a face $f_{sd,1}$ in SDC we find the reversal edge $reversal(e)$ and the face $f_{sd,2}$ that is adjacent to it. We then can add an edge to C connecting the nodes in C that are represented in SDC by the faces $f_{sd,1}$ and $f_{sd,2}$.

If all necessary connections are added in C we add the reversal edges and compute the faces of C using LEDA. We now copy all the edge attributes from SDC and calculate the vertex positions as described in section 1.2.3 on page 18.

The result is the Minkowski sum C of the convex polyhedra A and B.

3.10 Results

We can now test our algorithm using our own implementation. As mentioned in section 2.5 the (average) length and number of edges in a slope diagram of a polyhedron is fully dependent on the complexity of the polyhedron. This is affirmed by $\frac{I}{N_e}$ which is fairly constant. Therefore we can only vary the number of edges in our tests, and hence we only have a single table with results. In order to keep the results readable we split the table in two.

N	N_A	N_B	N_e	t_{naive}	$t_{overlay}$	t_{ph}	t_{ssa}
25	25	25	138	0.02	4.09	0.04	4.13
50	49	50	285	0.03	9.14	0.08	9.22
100	97	98	573	0.10	19.37	0.14	19.50
250	232	220	1,344	0.44	44.61	0.28	44.89
500	392	409	2,388	1.38	64.53	0.46	64.99
1000	624	655	3,680	3.43	91.16	0.62	91.78
2000	792	771	4,648	5.14	112.00	0.69	112.68
3000	790	831	4,831	6.18	109.42	0.69	110.11

Table 5: *Results of our implementation of the sphere sweep algorithm (part 1). N is the number of vertices generated for every slope diagram, before the convex hull is calculated. N_A and N_B are the number of vertices in slope diagrams SDA and SDB. N_e is the total number of edges in SDA and SDB. t_{naive} is the time needed to calculate the Minkowski sum of polyhedra A and B using the naive method. $t_{overlay}$ is the time needed to calculate overlay SDC of SDA and SDB. t_{ph} is the time needed to construct polyhedron C from its slope diagram SDC. $t_{ssa} = t_{overlay} + t_{ph}$.*

We ran the algorithm three times for every N and averaged the results.

N	O_1	O_2	O_3	$\frac{t_{ssa}}{O_3}$	$\frac{t_{naive}}{O_3}$	$\frac{I}{N_e}$	$\frac{t_{ssa}}{I}$	$\frac{t_{naive}}{I}$
25	981	448	1,429	2.89E-03	7.00E-03	4.57E-01	6.56E-01	1.59E-04
50	2,324	1,028	3,352	2.75E-03	8.95E-03	4.42E-01	7.31E-01	2.38E-04
100	5,250	2,181	7,431	2.62E-03	1.03E-03	4.15E-01	8.19E-01	3.22E-04
250	13,967	5,259	19,226	2.33E-03	1.82E-03	3.76E-01	8.87E-01	6.92E-04
500	26,797	9,639	36,436	1.78E-03	2.97E-03	3.60E-01	7.57E-01	1.26E-03
1000	43,591	14,309	57,901	1.59E-03	4.61E-03	3.28E-01	7.60E-01	2.21E-03
2000	56,624	15,947	72,571	1.55E-03	6.20E-03	2.82E-01	8.61E-01	3.44E-03
3000	59,122	16,142	75,264	1.46E-03	7.07E-03	2.73E-01	8.35E-01	4.03E-03

Table 6: *Results of our implementation of the sphere sweep algorithm (part 2). O_1 is calculated as $N_e \log N_e$, $O_2 = I \log N_e$, $O_3 = (N_e + I) \log N_e$.*

Every time we noted the time needed to calculate the Minkowski sum using the naive method (t_{naive}), the time needed to calculate the overlay of the slope diagrams of polyhedra A and B ($t_{overlay}$), and the time needed to construct the Minkowski sum from the overlay (t_{ph}). t_{ssa} is the sum of $t_{overlay}$ and t_{ph}.

What strikes us is the fact that the calculation of the Minkowski sum using our implementation of the sphere sweep algorithm requires significantly more time than the calculation of the Minkowski sum using the naive method. Nevertheless the table shows us that the sphere sweep algorithm runs in $O((N + I) \log N)$ time, as we could have expected.

Our application requires much time because the number of computations per iteration is quite large. Our implementation could be significantly faster with a few modifications to the code.

The most important information of the table is in the columns 8 and 9 of table 6. We compare the time per intersection for the two methods. This shows us that the sphere sweep algorithm requires constant time per intersection, whereas the naive method becomes less efficient when the number of edges in the slope diagrams grows.

We recollect the objectives of our work, as we stated them in our introduction. First of all, we wanted to design a new algorithm that calculates the Minkowski sum of two three-dimensional convex polyhedra. We aimed to make this algorithm more efficient than any existing method. Based on the known plane sweep algorithm we designed the sphere sweep algorithm that runs in $O((N + I) \log N)$, even slightly better.

Second of all, we wanted to make an implementation of our algorithm, so that we could test and analyze the algorithm. We ran our algorithm on pairs of randomly generated polyhedra, and compared the results with the Minkowski sums calculated using the naive method. The Minkowski sums

were exactly identical, which experimentally proves our algorithm to be correct.

Finally, we compared the time complexity of our algorithm with that of the naive method. Although the absolute running time of the sphere sweep algorithm is significantly worse than that of the naive method, we can conclude that the sphere sweep algorithm is more efficient. The last two columns prove us right. The time needed per intersection is linear for the sphere sweep algorithm, whereas it grows for the naive method.

3.11 Discussion

Although we have designed an algorithm that works correctly we can not present running times for our implementation that are significantly better than the running times of the naive method.

Some modifications, however, could give the time complexity of our implementation a boost.

First of all, our implementation is primarily slowed down by the use of rational numbers. In order to constrain their bit-length growth rational numbers are normalized after every calculation. We implemented our own normalization function, since it is not included in the version of LEDA we used. Fogel and Halperin [6] encountered a similar problem and noted that the use of another library like `CGal` might increase performance significantly.

Furthermore, we could gain significantly on the search and insert functions in the binary tree used for the status. For example, every search for neighbor segments costs $\log N$ time, with N being the number of segments in the status. This could be done in constant time if we stored references to left and right neighbors while inserting segments in the status.

Moreover, when we want to calculate the overlay of slope diagrams SDA and SDB we initialize our implementation by inserting all segments in one set. During the sweep algorithm segments are checked for intersections, independent of the original slope diagrams they belong to. A simple test whether they belong to the same slope diagram would make a check of two segments redundant.

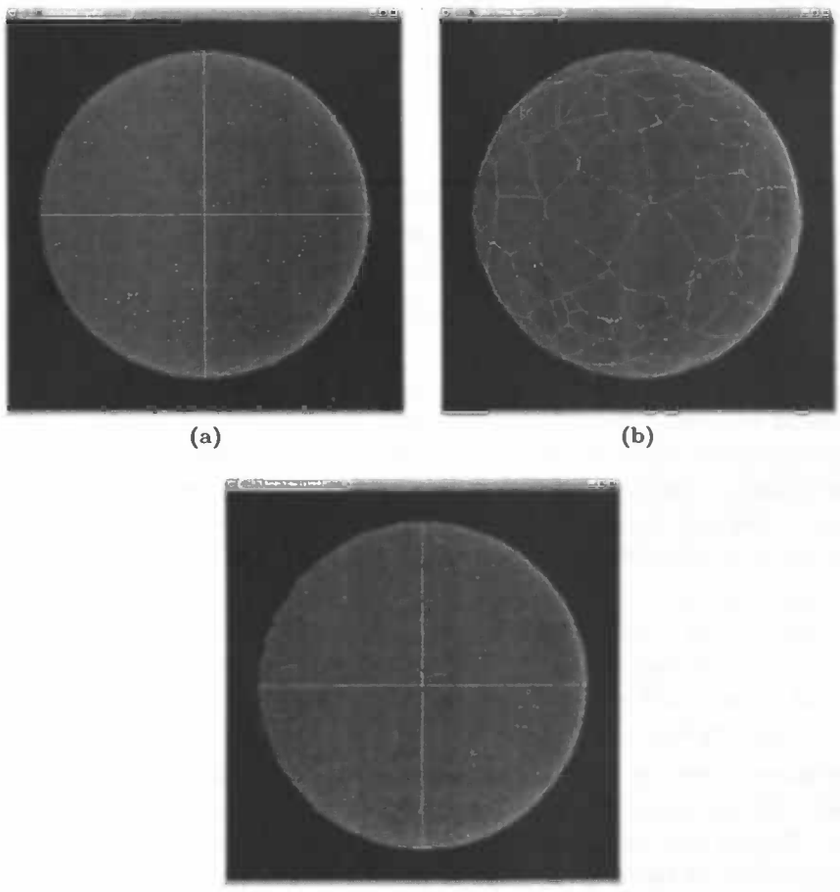


Figure 17: *Output of our sphere sweep algorithm implementation. Slope diagrams of a cube (a) and a polyhedron with randomly chosen vertices (b), and their overlay (c). Generated by our implementation of the sphere sweep algorithm*

References

[1] Henk Bekker and Jos B.T.M. Roerdink. An efficient algorithm to calculate the minkowski sum of convex 3d polyhedra. *Lecture Notes In Computer Science*, 2073:619 – 628, 2001.

[2] De Berg, Van Kreveld, Overmars, and Schwarzkopf. *Computational Geometry – Algorithms and Applications*, pages 20 – 29. Springer-Verlag, 1997.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 263 – 277. MIT Press/McGraw-Hill, 1990.

[4] K. de Raedt. Calculating the overlay of connected subdivisions on a sphere using a planar overlay algorithm. Master's thesis, Rijksuniversiteit Groningen, 2002.

[5] A. J. W. Duijvestijn and P. J. Federico. The number of polyhedral (3-connected planar) graphs. *j-MATH-COMPUT*, 37(156):523–532, 1981.

[6] Efi Fogel and Dan Halperin. Exact and efficient construction of minkowski sums of convex polyhedra with applications. To appear in Alenex 2006, 2006.

[7] Miguel Granados, Peter Hachenberger, Susan Hert, Lutz Kettner, Kurt Mehlhorn, and Michael Seel. Boolean operations on 3d selective nef complexes: Data structure, algorithms, and implementation. In *ESA*, pages 654–666, 2003.

[8] Kurt Mehlhorn and Stefan Näher. *LEDA: a platform for combinatorial and geometric computing.* Cambridge University Press, New York, NY, USA, 1999.

[9] R. Schneider. *Convex Bodies: The Brunn–Minkowski Theory.* Press Syndicate of the University of Cambridge, Cambridge, MA, 1993.

[10] A. V. Tuzikov, J. B. T. M. Roerding, and H. J. A. M. Heijmans. Similarity measures for convex polyhedra based on Minkowski addition. *Pattern Recognition*, 33:979–995, 2000.