DYNAMIC MINKOWSKI SUM OPERATIONS

by

Evan Behar
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Dr. Jyh-Ming Lien, Dissertation Director
_____	Dr. Amarda Shehu, Committee Member
_____	Dr. Dana Richards, Committee Member
_____	Dr. Walter D. Morris, Committee Member
_____	Dr. Sanjeev Setia, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Fall Semester 2016
George Mason University
Fairfax, VA

Dynamic Minkowski Sum Operations

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Evan Behar
Master of Science
George Mason University, 2010
Bachelor of Science
Rensselaer Polytechnic Institute, 2005

Director: Dr. Jyh-Ming Lien, Associate Professor
Department of Computer Science

Fall Semester 2016
George Mason University
Fairfax, VA

Table of Contents

List of Tables

List of Figures

Abstract

DYNAMIC MINKOWSKI SUM OPERATIONS

Evan Behar, PhD

George Mason University, 2016

Dissertation Director: Dr. Jyh-Ming Lien

The Minkowski sum is an important operation in a wide variety of applications, including robotic motion planning, computer animation, physical simulation, rapid prototyping, and computer-aided design, as well as being the fundamental operation in computing configuration spaces of objects. The computation of Minkowski sums focuses primarily on finite boundary representations. Although the Minkowski sum has been studied extensively since the 1970s, still the great bulk of work in the field of Minkowski sums focuses on computing the Minkowski sums of objects which do not rotate or deform, but remain in a single orientation and formation.

In many applications, however, the objects provided are transformed over time by means of rotation, scaling, or localized deformation, or make use of significantly similar or repeated geometry. In these cases it is often not practical to recompute the Minkowski sum after each transformation. This is largely owing to the computational complexity of the Minkowski sum–for non-convex objects in d dimensions, the worst-case boundary complexity of the Minkowski sums is $O(m^d n^d)$, where m and n are the number of facets in the inputs. However, because of the challenges of working with non-convex objects, the little existing work in computation re-use for Minkowski sums focuses solely on convex objects.

In computer graphics, for example, it is common to create a collection of similar objects by copying a master object and deforming or reorienting each copy in some way. To recompute the Minkowski sum of each such object from scratch is extremely expensive and undesirable. Instead, we would prefer to compute the Minkowski sum of the master object once, and then use the computation already done on the master object to update the sums for each such transformed copy efficiently. The main idea here is that when the transformation applied is small, the resulting change in the Minkowski sum boundary will typically also be small.

In this work, methods are presented for dynamically updating the Minkowski sum boundaries of non-convex triangle meshes global changes to the input mesh. The underlying Minkowski sum technique is based on the convolution of the meshes–a superset of the Minkowski sum boundary which is efficient to compute. The Minkowski sum boundary is then extracted from the convolution.

The methods proposed for rotation and scaling depend upon a dynamic convolution algorithm that operates on the Gauss map of P. The technique for local deformation relies on updating the convolution in only those regions affected by the deformation, limiting the amount of work necessary to extract the Minkowski sum boundary. These methods are implemented under the CGAL framework, and testing demonstrates that the computations are significantly faster than recomputing the Minkowski sum from scratch when the scale of the transformations is small, making these methods useful for applying the Minkowski sum to applications where the input operands must undergo transformations.

Chapter 1: Introduction

1.1 Motivation

The *Minkowski sum* of two point sets P and Q is defined by the set of pairwise sums of their points: $P \oplus Q = \{p + q \mid p \in P, q \in Q\}$. In particular, given polygons and polyhedra, we are interested in the Minkowski sums of these shapes. The set of polytopes is closed under the Minkowski sum, and so the Minkowski sum of two polygons will also yield a polygon, and similarly, the Minkowski sum of two polyhedra is itself a polyhedron.

Since the late 1970s, the Minkowski sum has been an important operation to researchers interested in a wide variety of fields. The Minkowski sum forms the fundamental operation for computing configuration spaces ($\mathcal{C}$-spaces) which are directly applicable to motion planning, virtual assembly and rapid prototyping tasks. Further, the Minkowski sum provides a straightforward way of performing tasks such as penetration depth estimation, offsets, rounding, and sweeping. It is also used for continuous collision detection, physical simulation and mathematical morphology operations such as dilation and erosion.

However, research into the Minkowski sum slowed significantly in the 1980s, and continued to be sparse until very recently due to the high computational complexity of the Minkowski sum boundary for general inputs and the intractability of computing geometric boundaries in high dimensional spaces. Given two non-convex d-dimensional inputs of boundary complexities m and n, the worst-case boundary complexity of the Minkowski sum boundary is $O(m^d n^d)$.

Even explicit boundary computations for rotating polygonal inputs, whose boundaries are only as complex as those of non-rotating polyhedral inputs, have mostly eluded researchers until recently [9] . However these explicit representations remain important. In virtual

assembly and rapid prototyping, motion planning problems that require incomplete planners, continuous collision detection, packing problems, and penetration depth estimation algorithms, efficient computation of the Minkowski sum remains a the rate limiting step.

The algorithms in existing work have mostly centered on computing *static* Minkowski sums of polygons and polyhedra. These algorithms are generally unsuitable for use in applications where such inputs must transform in some ways, as the boundary must be recomputed from scratch with every transformation of either input.

1.2 Complexity analysis and dynamic operations

Table 1.1 summarizes worst-case complexity of the Minkowski sum boundary in both two and three dimensions, for inputs P and Q of input size m and n respectively.

Table 1.1: Worst-case boundary complexity of the Minkowski sum for given input types

	convex $\oplus$ convex	convex $\oplus$ non-convex	non-convex $\oplus$ non-convex
polygons	$O(m+n)$	$O(mn)$	$O(m^2n^2)$
polyhedra	$O(mn)$	$O(m^2n^2)$	$O(m^3n^3)$

As shown in the table above, for general polygons and polyhedra, the worst-case boundary complexity is quite high, such that even worst-case optimal algorithms for computing the Minkowski sum can be quite slow in practice.

In many applications, it is desirable for at least one of the inputs to transform in some way. Perhaps the most common example is rotation of a rigid body. This is a common input change in virtual assembly. As parts are developed separately and manipulated together, the addition of rotational velocity is frequently a component of collision response. Packing problems can be better served by taking into account degrees of freedom of rotation, and robots in motion planning problems may need to change orientation in order to better

navigate through an environment.

Given the high complexity of the Minkowski sum boundary, it is undesirable to discard all existing computation and simply recompute the boundary of the transformed inputs from scratch. Ideally, we would like to take advantage of computation we have already done in computing the Minkowski sum boundary and only recompute portions which cannot be updated.

Most existing strategies for addressing this problem either enumerate all of the structural changes that may occur with regard to a particular degree of freedom (dof), such as a single rotation axis [26], or approximate these changes via discretization of the dof [42]. These strategies are generally not practical for situations in which the inputs possess many dofs, and so we are interested in the idea of dynamically updating the Minkowski sum boundary.

In this work I address a number of strategies for computing Minkowski sums dynamically across a variety of changes to the input models. In Chapters 3 and 4, I address strategies for computing the Minkowski sums of polygons in two dimensions. In Chapter 5, I provide a method for dynamically updating the Minkowski sum of convex polyhedra. In Chapter 6, I provide methods for the scaling of general polygons and convex polyhedra. In Chapter 7, I introduce a dynamic convolution algorithm, which may be used to allow dynamic Minkowski sum methods for convex polyhedra to be used on non-convex inputs, and discuss a new continuous penetration depth algorithm based on the convex map. In Chapter 8, I analyze the results of my research and discuss future work.

Chapter 2: Related work

The work in this dissertation builds on related work in a number of fields: numerical robustness, which allows us some guarantee on the exactness of our computations; the notion of the arrangement, which allows us to create non-overlapping regions from a set of potentially overlapping line segments; the notion of a convolution of polytopes, from which Minkowski sum boundaries can be extracted; and previous methods of computing Minkowski sum boundaries. In this chapter, we give an overview on each of these topics.

2.1 Numerical robustness

Because we demonstrate algorithms which can compute Minkowski sum boundaries exactly, it is important to discuss the issue of numerical robustness and models of robust computing before we proceed further. Because geometric computations take place over the domain of real numbers and there are many arithmetic computations necessary in order to determine such things as intersections between primitives, numerical error when using floating point types is inevitable.

However, in geometric computing, numerical error can easily introduce significant errors into results. For example, numerical error in computing the intersections between line segments can result in the wrong regions being identified in the result. If these regions represent important information, such as the location of a building on a map, such errors can result in errors which render the result unusable for its application. Further, numerical error can result in important properties of groups such as closure failing to hold. There are several different proposed strategies for dealing with numerical robustness, which we address briefly below.

2.1.1 Arbitrary precision floating point representations

Arbitrary precision floating point types attempt to address rounding errors by allowing the introduction of additional bits of precision at run-time. Given certain assumptions about the input, it may be possible then to determine the maximum number of bits of precision required such that even with rounding errors, ordering predicates will evaluate correctly, or that the results of computation can be considered to be exact. For example, in computing the determinants of $n \times n$ matrices, a precision of $nL + n \lg n$ bits is sufficient when the entries are L-bit integers. [52] Arbitrary precision types suffer from efficiency concerns due to requiring special logic not necessary with built-in floating point types. Additionally, there may be extra slowdown incurred in detecting precision problems and adjusting the available precision at run-time. Also, while there are a number of arbitrary precision packages in C and C++ to perform arbitrary precision arithmetic, such as the GNU GMP[1] and MPFR[2] projects, overall, the theoretical underpinnings of selecting optimal precisions are not well-explored.

Benouamer, et. al. [14] describe a lazy method for updating the precision of an arithmetic expression in which an evaluation tree is constructed for the predicate and a desired precision is set for the root node. They then iteratively update the precision at the leaf nodes until the desired precision is obtained for the root node. Yap and Dubé [52] in contrast propagate the root's precision to all of its leaves automatically, computing the leaves using primitives to approximate the result to the given precision. However, both of these methods accept the precision of the root node as a user input and do not explore in detail how to choose a satisfactory precision.

2.1.2 Exact computation

Yap and Dubé [52] also defined the *exact computation paradigm*. The exact computing paradigm as described strives to render numerical errors nonexistent by representing a closed subset of the real numbers exactly via algebraic representations. Examples of exact

[1] GMP - http://gmplib.org/
[2] MPFR - http://www.mpfr.org/

computing types are rational number types and square-root extensions. Rational types store a numerator and a denominator rather than computing a fixed or floating point representation explicitly. Square root extensions extend the rationals into other closed subsets of the reals, being expressed in the form $a + b\sqrt{r}$, where a, b and r are all rational numbers. Square root extensions are closed for values with the same root value r, and can be efficiently compared to each other even when the roots differ. However, computation under exact types is subject to overflow constraints when using built-in integer types and inefficiency when using custom "big-int" numerical types.

They describe a further extension from simple square root types using *isolating interval representations*. Given a polynomial $P(x)$ with real root α, it is possible to represent α as the pair $(P(x), I)$ where I is an interval in $\mathbb{R}$ with rational endpoints such that I contains no other real roots of $P(x)$. This representation, like many exact representations, suffers from the property that it is not unique.

2.1.3 k-th order statistics

In computing the k-way union of polygons, Lu, et. al. [43] propose a robustness model for finding the union boundary of polygons that eschews computing intersection points entirely. Instead, the closest intersection is determined by using visibility metrics. Given a point q on a segment s, and a line segment t in the set of polygon edges $\mathcal{S}$, the visibility between q and t is determined by sign$(\overrightarrow{qr} \cdot \mathbf{t_n})$ where r is a point on t and $\mathbf{t_n}$ is the normal of t. If sign$(\overrightarrow{qr} \cdot \mathbf{t_n}) > 0$, then q and t are visible to each other, if it is 0, then q is on t, and if it is less than 0, then q is invisible to t.

The algorithm proceeds by partitioning $\mathcal{S}$ into sets of visible, invisible, and "on" segments for a given q, denoted $\mathcal{V}, \mathcal{I}, \mathcal{O}$ respectively, and iteratively selecting a new q until $|\mathcal{I}| = 1$, in which case $s \ni q$ is the segment with the intersection we're searching for. As compared to computing intersections explicitly, which requires many arithmetic operations, each visibility operation computes only a dot product, which consists of only two multiplications and one

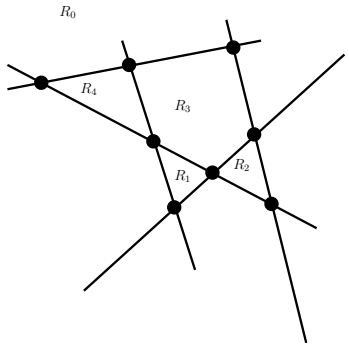


Figure 2.1: A sample arrangement of line segments in a plane. R_0 is the unbounded outer region, while R_1 through R_4 are the internal bounded regions. The black circles represent the intersections of the line segments

summation. As a result, accumulated error from mathematical operations is significantly smaller.

The algorithm is also able to determine when standard floating point or double precision types are insufficient. If the search range for the desired segment is empty but there are still segments left in $\mathcal{S}$, then the segments of $\mathcal{S}$ cannot be adequately distinguished at the current precision and the precision is doubled to enable the computation to continue. In computing the intersections explicitly, determining adequate precision is not straightforward.

There are two major drawbacks of the k-th order statistics method. First, it only applies to finding intersections between primitives and has no current extensions to other geometric tasks. Secondly, it is currently only well-defined in two dimensions. The visibility concept has not been extended to three dimensions and is thus not useful for polyhedra at the moment.

2.2 Arrangement

Having discussed the underlying numerical robustness for geometric computations, it is also important to discuss briefly the arrangement of primitives. Given a set of primitives P in $\mathbb{R}^d$, the arrangement of $P =< I, S, R >$, where I is the set of all intersections between primitives of P, S is the set of all sub-divided primitives formed by those intersections, and R is the set

of *regions* enclosed by I and S. Figure 2.1 demonstrates an arrangement of line segments in a plane.

The arrangement is important in computing the Minkowski sum for several reasons: many union techniques depend upon computing the arrangement of the inputs, which is important in convex decomposition methods, defined and discussed in Section 2.4.1. Also, the arrangement is used heavily in the convolution-based methods defined and discussed in Section 2.4.2.

Edelsbrunner's zone theorem [22] demonstrates that the output complexity of the arrangement is $O(n^2)$ for n line segments. He also demonstrates that the arrangement may be computed in worst-case $O(n^2)$ time by adding line segments incrementally.

2.3 Convolution

In 1983, Guibas et al. [30] propose the idea of the convolution as part of a kinetic framework for geometric computing. They defined the notion of polygonal tracings, curves in a plane which have both a direction of motion and a well-defined outward normal direction. They further defined the notions of convex tracings. The convolution of polygonal tracings has the property that, given two totally convex polygonal tracings as input, their convolution is the same as the boundary of their Minkowski sum, and that for non-convex tracings, the convolution is a superset of the Minkowski sum boundary. Since then, Basch et al. [8] have extended the notion to polyhedral tracings, and the convolution has become an important underlying operation in computing the Minkowski sums of shapes–itself a fundamental operation in many problems.

In 1993, Ghosh [28] propose the use of a *slope diagram* and an operation called the *boundary sum*. The slope diagram is essentially equivalent to the Gauss map from which the convolution is computed, at least in the sense that one can compute the Gauss map trivially from the slope diagram and vice versa, however the boundary sum that Ghosh defines is actually a superset of the convolution, which is, itself a superset of the Minkowski sum

boundary.

However, since [8] there has not been much development on computing convolutions. In particular, while there has been an increase in interest for computing dynamic Minkowski sums, the dynamic computation of convolutions has not generally been raised as an issue except in the computation of Minkowski sums of convex objects in our own previous work [10, 13] where the convolution and the Minkowski sum are identical.

2.4 Computing Minkowski sum boundaries

In order to properly address the related work, it is important to have a solid understanding of the foundational ideas behind computing the boundary of the Minkowski sum.

Problem statement 1. *Let $\partial(P)$ denote the boundary of a shape P. Given inputs $\partial(P)$ and $\partial(Q)$, we wish to compute $\partial(P \oplus Q)$.*

Existing methods for computing the boundary of the Minkowski sum can broadly be divided into two paradigms, *convex decomposition*-based methods, and *convolution*-based methods. *Convex decomposition*-based methods, as the name implies, decompose the inputs into convex pieces and then compute pairwise Minkowski sums for these convex pieces. In order to construct the final Minkowski sum boundary, these pairwise sums must then be unioned together.

In contrast, *convolution*-based methods use the idea of *compatibility* between the primitives of the inputs to construct the convolution of the inputs, which is known to be a superset of the Minkowski sum boundary []. The convolution boundary must then be trimmed in order to obtain the final Minkowski sum boundary.

2.4.1 Convex decomposition

Formally, given boundary-defined input shapes P and Q, convex decomposition methods decompose P and Q into sets of disjoint convex shapes, $P_{convex} = \{P_1, P_2, ..., P_m\}$ and

$Q_{convex} = \{Q_1, Q_2, ..., Q_n\}$. The set of all pairwise sums, $S_{pair} = \{P_i \oplus Q_j \mid 1 \leq i \leq m, 1 \leq j \leq n\}$, is computed and the final Minkowski sum boundary is computed as $\partial(\bigcup_{s \in S_{pair}} s)$. This takes advantage of the fact that computing the Minkowski sum of convex shapes is much more efficient than that of non-convex shapes.

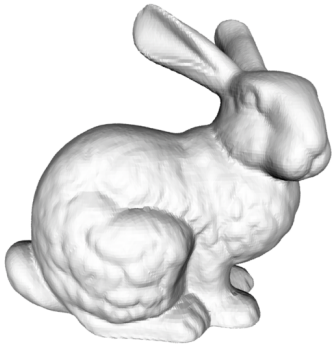
Convex decomposition is a popular method because of the conceptual simplicity of the approach. However, the convex decomposition strategy has significant drawbacks. In particular, for polygons with holes as well as for general non-convex polyhedra, finding an optimal convex decomposition of the input shape is known to be NP-hard [47].

Convex decompositions are also not unique, and Agarwal, et. al. [1] determined that the decomposition strategy chosen greatly influences performance. The number of convex components may be very large, resulting in a potentially huge number of pairwise Minkowski sums that must be computed and then unioned together. In Figure 2.2, the model of the bunny was decomposed into 16549 convex components, the model of the David into 85132 convex components. This results in over 1.4 billion pairwise Minkowski sums.

The performance of the union step is also heavily dependent upon the order in which these unions are computed, as traditionally the union operator is implemented as a binary operation. Recently, k-way polygon unions have been produced [43], but these methods have not yet been extended to polyhedra. In general the union operation is difficult to implement both efficiently and robustly, due to the many possible degeneracies.

Additionally, because the union operation is difficult for open sets, features of the Minkowski sum boundary may be lost during the union step due to regularization. The regularized union of sets S and T, denoted $S \cup^* T$, is defined as the closure of $S \cup T$. The regularized Minkowski sum of S and T is $\partial(S \oplus T) \cup^* (S \oplus T)$, and we denote it $S \oplus^* T$.

As shown in Figure 2.3, some non-manifold portions of the sum boundary may represent narrow passages in motion planning and other problems requiring the contact space of the inputs. Thus, great care must be taken when using convex decomposition for these applications since completeness of the result may be compromised.



(a) bunny, 16549 convex components

(b) david, 85132 convex components

Figure 2.2: For these inputs under the convex decomposition paradigm, the total number of pairwise Minkowski sums needing to be unioned is over 1.4 billion.

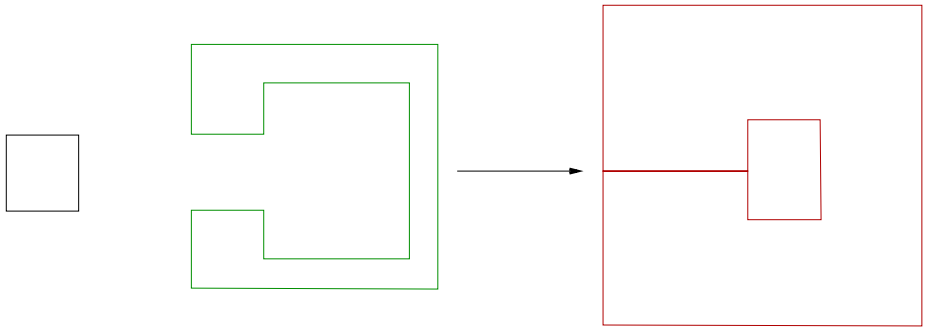


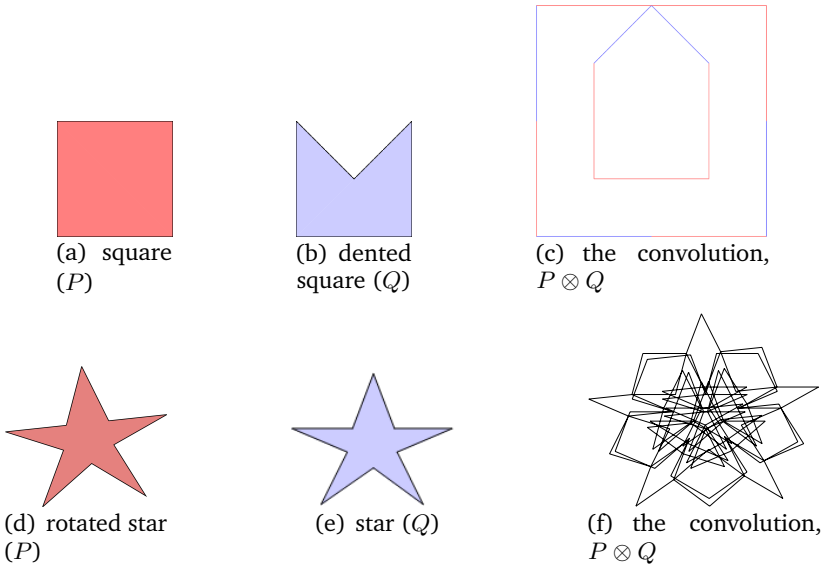
Figure 2.3: (left) P, (center) Q, (right) $\partial(P \oplus Q)$, demonstrating a non-manifold boundary edge in the Minkowski sum boundary which represents a narrow passage.

Keeping an eye toward the proposed work, convex decomposition strategies are difficult, even conceptually, to map to an idea of dynamically updating the Minkowski sum. A large quantity of intermediate geometry would be necessary in order to avoid recomputing the entire decomposition and union steps, which is impractical for complex inputs.

2.4.2 Convolution

Computing the Minkowski sum boundary is closely related to another geometric operation, the *convolution* of P and Q, denoted $P \otimes Q$. The convolution of P and Q is defined using the notion of *compatibility* between primitives: vertices, edges and faces. The Minkowski sums of so-called compatible primitives form the boundary of the convolution, which is known to be a superset of the Minkowski sum boundary. In particular, when P and Q are convex, $P \otimes Q = \partial(P \oplus Q)$.

Convolution-based methods work by computing $P \otimes Q$, and then extracting the Minkowski sum boundary by removing superfluous portions of the convolution. If we consider the sets defined by P and Q to be the open sets defined by their boundaries, rather than their closures, then the Minkowski sum $P \oplus Q$ is the open set defined by $\partial(P \oplus Q)$, and extracting the Minkowski sum boundary is tantamount to identifying the maximal subset S of the convolution such that S is also a subset of the Minkowski sum: $S \subset P \oplus Q$ and $P \oplus Q - S \cap P \otimes Q - S = \emptyset$.



(a) square (P)

(b) dented square (Q)

(c) the convolution, $P \otimes Q$

(d) rotated star (P)

(e) star (Q)

(f) the convolution, $P \otimes Q$

Figure 2.4: Two sample polygon convolutions

Because identifying S constitutes the great bulk of the work done in convolution-based methods, we defer this discussion to specific methods later in the chapter. In the following sections, we define and demonstrate the notion of compatible primitives and the convolution boundary as preliminaries. A more detailed survey of techniques for extracting the Minkowski sum boundary from the convolution is presented in Sections 2.5 and 2.6.

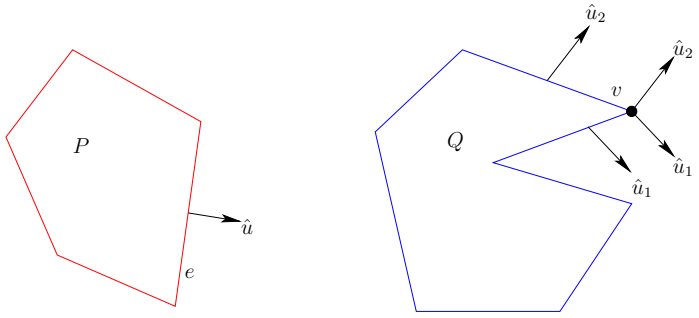


Figure 2.5: Two polygons, P and Q. Labeled: an edge e of P, a vertex v of Q, and the edge normals $\hat{u}$, $\hat{u}_1$, and $\hat{u}_2$.

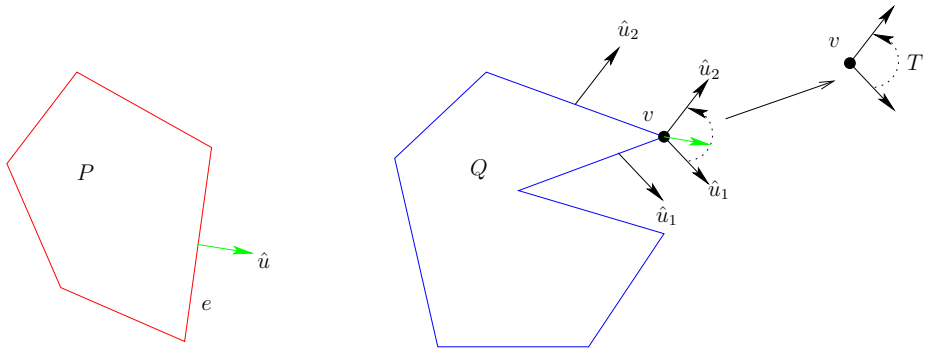


Figure 2.6: The dashed arc shows the turn. In the upper-right, we isolate v, $\hat{u}_1$, $\hat{u}_2$ and T. Since $\hat{u}$ is within the arc formed by the turn, $e \oplus v$ will be an edge of the convolution.

Compatibility of polygons in a plane For polygons, the idea of compatibility only applies between an edge e of P and a vertex v of Q, or vice versa. As defined above, when e and v are compatible, the edge $e \oplus v$ is contributed to the convolution boundary–we say that v *convolves* with e, and vice versa. To determine compatibility between edges and vertices, we consider the outward normals of the edges of P and Q.

Without loss of generality, we assume that P and Q are oriented counter-clockwise. Consider an edge e of P with outward unit normal $\hat{u}$, and a vertex $v = v_n$ of Q such that v has incident edges, $e_1 = (v_{n-1}, v)$, $e_2 = (v, v_{n+1})$ with outward unit normals $\hat{u}_1$ and $\hat{u}_2$ respectively, as illustrated in Figure 2.5.

We denote the counter-clockwise rotation of $\hat{u}_1$ by some angle θ as $R(\hat{u}_1, \theta)$, and define $\theta_p < 2\pi$ to be the angle such that $R(\hat{u}_1, \theta_p) = \hat{u}_2$.

Definition 1. *We call the clopen interval $T = [0, \theta_p)$ the* **turn** *between e_1 and e_2, (identically, the turn of v), and we say that the unit normal $\hat{u}$ of e lays on this turn iff there exists some $\theta \in T$ such that $R(\hat{u}_1, \theta) = \hat{u}$,*

See Figure 2.6 for an example. Having defined the turn, the notion of the compatibility of v and e follows directly from it.

Definition 2. *We call v and e* **compatible** *iff e's outward unit normal $\hat{u}$ lays on the turn of v.*

This is a fairly straightforward application of the turn. In Figure 2.6, e is compatible with v since $\hat{u}$ lays on the turn of v.

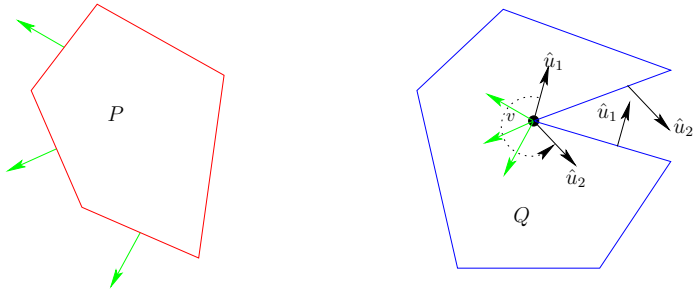
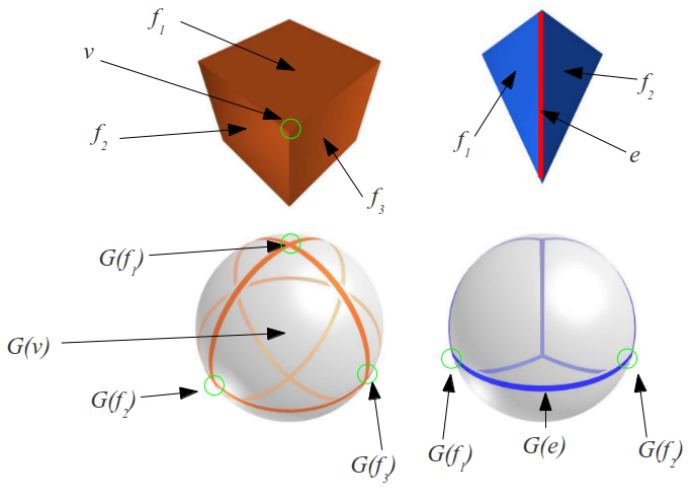


Figure 2.7: The dashed arc shows the turn. Each edge of P marked with a green outward normal will convolve with v. Notice that the turn of the reflex vertex here is much larger than that of the convex vertex in Figure 2.6.

It is important to note that because the turn is defined according to a counter-clockwise rotation of e_1, when v is a reflex vertex, that is, a vertex whose interior angle is greater than π as in Figure 2.7, the length of the turn interval will be greater than π. As a result, reflex vertices can convolve with potentially many more edges than convex vertices.

Compatibility of polyhedra In three dimensions, P and Q are polyhedra, and the notion of the turn between edge normals does not adequately describe the relationship between primitives. In order to extend this, we turn to the notion of the *Gaussian map*.

The Gaussian map is a mapping of the surface normals of the faces of a polyhedron to points on the surface of a unit sphere. Examples are shown in Figure 2.8 for a cube and a tetrahedron. We denote the Gaussian map of P as $G(P)$ (similarly for Q as $G(Q)$). Similarly, we denote the mapping of a primitive p as $G(p)$. In Figure 2.8, The faces f_1 and f_2 map to the points shown on the surface of the unit sphere as $G(f_1)$ and $G(f_2)$. The vertex v in the first example corresponds to the region on the surface of the sphere marked $G(v)$, and the edge e which is incident to both f_1 and f_2 maps to a geodesic arc $G(e)$, which has $G(f_1)$ and



Images adapted from *Minkowski Sums of Convex Polyhedra*, Efi Fogel, 2009

Figure 2.8: Gaussian map of a cube (left) and a tetrahedron (right). The edge e highlighted in red on the tetrahedron maps to the geodesic arc $G(e)$ in the Gaussian map.

$G(f_2)$ as its endpoints.

Strictly speaking, the Gaussian map does not preserve information about the primitives which are used to create the map. Given P and $G(P)$, it not necessarily possible to determine that the face which originated the point $G(f_1)$ is in fact f_1, only that $G(f_1)$ is the normal of some face in P. Because of this, convolution methods typically employ an *extended Gaussian map*, under which $G(f_1)$ is labeled with its originating face. For the remainder of this proposal, when we refer to the Gaussian map we will be talking about the extended Gaussian map.

It is important to note that since the geodesic on the unit sphere consists of the great circle containing both $G(f_1)$ and $G(f_2)$,there are actually two arcs along the geodesic which connect these points. $G(e)$ will be the shorter arc between $G(f_1)$ and $G(f_2)$ when e is a convex edge, and the longer arc when e is a reflex edge.

To see that this is the case, we can project f_1 and f_2 to line segments in the plane of the geodesic which share a vertex $proj(e)$ which is projected from the edge e, as in Figure 2.9. This projection preserves the orientation of the faces and their incident edge. As a result,

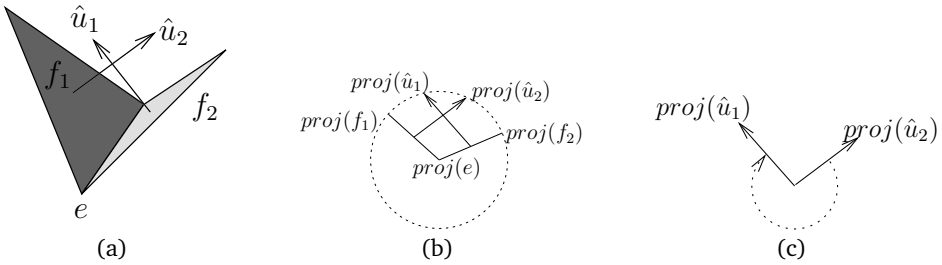


Figure 2.9: (a) The faces f_1 and f_2 adjoin a reflex edge e. (b) The projection to the plane of the Gaussian map's geodesic. (c) The turn of $proj(e)$.

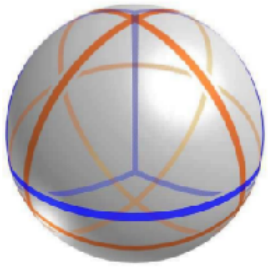


Figure 2.10: Overlay of the Gaussian maps for the cube and the tetrahedron

if e is a convex edge, the turn of $proj(e)$ will be precisely the shorter arc of the geodesic. Similarly, if e is a reflex edge, the turn of $proj(e)$ will be the longer arc of the geodesic.

Having defined the Gaussian map, the compatibility of primitives in P and Q arises from their *overlays*. Figure 2.10 shows the overlay for the cube and the tetrahedron. We define two distinct types of compatibility in three dimensions: *vertex-face (vf)* compatibility, and *edge-edge (ee)* compatibility. A vertex v from P and a face f from Q (or vice versa) are compatible iff $G(f)$ is inside the region formed by $G(v)$ in the overlay of the Gaussian maps. Notionally similar, two edges, e_P from P and e_Q from Q are compatible iff $G(e_P)$ and $G_(e_Q)$ intersect in the overlay of the Gaussian maps, as shown in Figure 2.11. The overlays are computed as the arrangement of the geodesic arcs in the Gaussian maps.

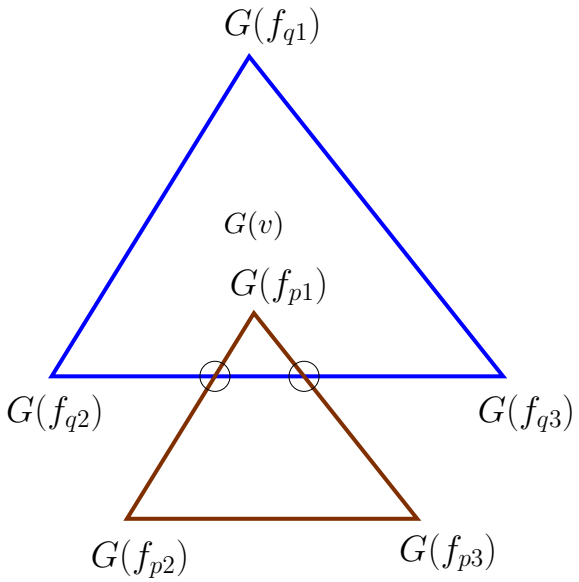


Figure 2.11: A flattened region of a Gaussian map overlay. In this example, f_{q2} is compatible with v_p because $G(f_{q2})$ is in the region of $G(v)$. Additionally, the edge between f_{q2} and f_{q3} is compatible with the edge between f_{p1}, f_{p2} because $(G(f_{q2}), G(f_{q3}))$ intersects with $(G(f_{p1}), G(f_{p2}))$ in the overlay.

Other representations

A small number of works have attempted to compute the Minkowski sum boundary by eschewing the standard paradigms. These include point-based approximations of the boundary [38, 46], voxelization approaches [36, 49], and a linear programming technique for d-dimensional polytopes [27]. We discuss these approaches briefly below.

2.5 Minkowski sum methods for convex inputs

Computing the Minkowski sum of convex shapes is drastically more efficient than computing those of non-convex inputs as shown in Section 1.2, and is an integral part of the convex decomposition strategy for computing Minkowski sums of non-convex shapes. Due to its relative simplicity, fewer works concentrate strictly on static Minkowski sums of convex shapes.

Lozano-Perez [42] proposed a method for computing $\partial(P \oplus Q)$ based on the idea of sliding Q around the surface of P. The algorithm selects a reference vertex on the boundary of Q, and then computes contact configurations of Q such that at least one vertex of Q is in contact with a vertex of P. To compute the final Minkowski sum boundary, edges connect the positions of Q's reference vertex as it traverses the edges of P. Because the sliding method must compute contact configurations for P and Q, it runs in $O(mn)$ time, much worse than the $O(m+n)$ worst-case complexity of convex P and Q.

In the same paper, another method was introduced which approaches the notion of the convolution. The method functions by inducing a total ordering on the normals of the edges in P and Q according to their rotation angles from a pre-determined unit vector $\hat{u}$. It then merges the edges based on this total ordering. This is tantamount to the merge step of the merge-sort algorithm, as the insertion of each contributed edge to the Minkowski sum boundary takes constant time, and so this merge-based algorithm runs in $O(m+n)$ time.

In 2004, Fukuda [27] introduced a linear programming solution to construct the Minkowski sum of d-dimensional convex polytopes. Fukuda's method depends on the fact that when P and Q are convex, $P \oplus Q = conv(vert(P) \oplus vert(Q))$, where $vert(M)$ is the set of vertices on a model M and $conv(S)$ is the convex hull of S. Fukuda computes the Minkowski sum of just the polytope vertices, and then induces a graph on it. He then constructs a spanning tree on the graph using linear programming to locate the extreme points of the graph and construct the convex hull. The linear programming problem is solved using the simplex algorithm, and runs in $LP(d, \delta)$, where d is the dimensionality of the inputs, and δ is the maximum degree of the induced graph. While this is the only known efficient algorithm for polytopes of arbitrary dimensionality, the complexity of the linear programming problem renders it generally unsuitable for polygons and polyhedra.

More recently, Fogel and Halperin [26] introduced a method for computing the Minkowski sum of convex polyhedra using the notion of a cubical Gaussian map. The cubical Gaussian maps surface normals of the polyhedra faces to the surface of a cube, instead. The cube can then be unfolded into six bounded planar regions, and the arrangement of the overlay on

each face of the cube can be computed using standard planar arrangement methods, which are simpler to compute than the arrangement of the standard Gaussian overlay.

2.6 Minkowski sum methods for non-convex inputs

Due to the greater complexity and generality of methods for computing Minkowski sums of non-convex inputs, there is a significantly larger body of existing research on Minkowski sums of non-convex (general) inputs. We subdivide this section into methods for polygons, and methods for polyhedra.

2.6.1 Minkowski sums of non-convex polygons

We begin again with Lozano-Perez [42], who extended the merging algorithm to non-convex shapes by surface decomposition of the polygon into *convex arcs*. These convex arcs are then closed into polygons. The result is, fundamentally, a convex decomposition scheme where the convex decomposition is performed by boundary decomposition instead of solid decomposition. However, there is no union step–the computed boundaries are simply reported all together, leaving self-intersections in. As a result, the time complexity of the method is $O(mn)$, but it does not provide a proper Minkowski sum boundary.

In 1997, De Berg, et. al. [19] proposed to use triangulation of polygons as a straightforward method of decomposition, since polygons may be triangulated efficiently. Triangulation of a polygon yields $O(n)$ pieces regardless of the input geometry, and so we must merge $O(mn)$ pairwise Minkowski sums for two non-convex inputs of size m and n respectively. The approach is straightforward, but the linear number of convex components is quite high, and so the union step must produce a comparatively large number of intermediate geometries. Even though the worst-case complexity of the algorithm is $O(m^2n^2)$, which is worst-case optimal, in practice this method is relatively slow compared to later methods.

In particular, in 2006 Wein [50] proposed to use a convolution-based method to compute

the Minkowski sum boundary. Wein's algorithm first computes $P \otimes Q$, and then its arrangement. The method then computes the *winding number* of each cell in the arrangement. The winding number of a point p with respect to a closed curve $\mathcal{C}$ is defined to be the number of full counter-clockwise rotations $\mathcal{C}$ traces around p, less the number of full clockwise rotations $\mathcal{C}$ traces around p. Any point in a given cell in the arrangement of the convolution will have the same winding number, so it is trivial to extend the notion of the winding number from the point to the cell. After computing the winding number of every cell, Wein's algorithm removes any edge in the arrangement which is not adjacent to a cell with winding number 0. Wein's method is currently the default algorithm for polygons used by CGAL[3]

The worst-case complexity of the convolution is $O(mn)$, since at most, each of the m edges in P can convolve with n vertices in Q and vice versa. Consequently, computing the arrangement of the convolution is worst-case $O(m^2n^2)$. The winding number of the cells in the convolution can be computed in linear time, as can removing edges from the arrangement. As a result, Wein's algorithm is worst-case optimal. Moreover, Wein's method outperforms convex decomposition strategies except in rare cases where (1) optimal or near-optimal convex decompositions of the inputs are extremely easy to compute, and (2) the complexity of the resulting Minkowski sum exhibits worst-case behavior.

2.6.2 Minkowski sums of non-convex polyhedra

In 1993, Ghosh [28] introduces the idea of the slope diagram, which is similar to the Gaussian map. There are two major differences between the slope diagram and the Gaussian map. First, the slope diagram stores edge length information from its originating faces. As a result, the originating model can be reconstructed up to scale strictly from the slope diagram. Secondly, the slope diagram eschews the use of "turns", instead adopting the concept of *sense*. The sense of a face of a polyhedron (edge of a polygon) is positive if it is adjacent only to convex edges (vertices), and negative if it is adjacent to at least one reflex edge (vertex). In the slope diagram, adjacent primitives are always connected by the shorter arc of the

[3]The Computational Geometry Algorithms Library. http://www.cgal.org

geodesic, but these arcs are annotated with sense information.

Ghosh defines a boundary sum, $P \uplus Q$, which is practically identical to the convolution $P \otimes Q$ except that it uses the slope diagram to construct its compatibility instead of the Gaussian map. Because of the slope diagram convention of always using the shortest arc between mapped primitives, the boundary summation is nominally different in structure from the convolution unless P and Q are both convex. Ghosh provides an algorithm for computing $P \uplus Q$, but does not provide any pruning algorithms for extracting the Minkowski sum boundary when the inputs are non-convex.

Varadhan and Manocha [49] propose to use a convex decomposition strategy. Their approach is novel in that instead of computing the union of the pairwise sums, they instead voxelize the space and extract an isosurface from the space of pairwise Minkowski sums by using an improved marching cubes algorithm. The approximate boundary they produce is fairly accurate and guaranteed to yield the same topology as the exact Minkowski sum, however their method does not tolerate degeneracies in the pairwise sums.

Lien proposes a point-based method [37] and a convolution-based method [38] for computing the Minkowski sum boundary. The point-based approximation uses the notion of a d-covering on the input: a sampling of points such that given any point s in the sampling, there exists another sample point t such that $dist(s,t) \leq d$. The algorithm computes a d-covering on both P and Q, and then computes the Minkowski sum of those covers. The resulting point set is a superset of a d-cover of the actual Minkowski sum boundary. Superfluous points are filtered out by removing points whose originating primitives are not compatible. Then the neighborhood of the boundary is identified using an octree; points not in this neighborhood are rejected. Finally, the points in the neighborhood of the boundary are culled using collision detection.

The convolution-based method in [38] constructs a superset of the convolution facets (owing to numerical error, rather than any particular strategy for extracting the boundary), and the intersections of the convolution facets with each other. Then, for each face, the planar arrangement of the face with its intersections is computed. The cells in these planar

arrangements are culled using collision detection tests. Though the method is quite efficient, it does not compute the exact Minkowski sum boundary–it may identify some boundary regions as superfluous which represent tight passages, sacrificing accuracy for speed.

More recently, Peter Hachenberger proposed a new decomposition strategy for polyhedra based on the idea of *sight walls*. A sight wall of a reflex edge e is the set of all points which can be connected to e by a vertical line segment without intersecting a face, edge or vertex. These sight walls are very effective at decomposing polygons, yielding in the worst case $O(r^2)$ convex pieces, where r is the number of reflex edges in the input. Hachenberger's method is the decomposition strategy implemented in CGAL.

In 2011, Barki [7] proposed a method for a non-convex/convex pair of inputs based on the idea of contributing vertices. The idea behind contributing vertices is that they are vertices of one input which are "compatible" with a face f of the other input by virtue of being at a maximal distance from the plane on which f lies. However, the method of contributing vertices put forth here only works when one of the input is convex, thus limiting its utility.

In 2015, Zhang and Zheng [54] presented a method for extracting the Minkowski sum boundary of general polyhedra based on [7] by generalizing the contributing vertex method to polygons. The runtime of their method is comparable to that of the work that we present in [12]. Also in 2015, Baram, et al. [6] proposed an exact method for the Minkowski sum of general polyhedra based on the reduced convolution, which we introduce in Chapter 3. This method operates by finding holes in each input that are "small" relative to the other input and filling them, which reduces the complexity of the inputs and therefore the time complexity of finding the Minkowski sum.

2.7 Dynamic Minkowski sum operations

At present there are only two published works related to the subject of dynamic Minkowski sums. Mayer, et. al. [45] deal with the rotation of convex polyhedra by computing a *criticality*

map. When rotating an input polygon P by some angle θ about a fixed axis, the underlying structure of the Minkowski sum will change as P rotates. However, the structure does not change continuously–these structural changes only occur at a finite set of *critical values* of θ. Mayer, et. al., take advantage of this to compute a data structure, the criticality map, which stores these critical θ values, along with a template of the Minkowski sum structure at the critical value.

The criticality map is set up for fast retrieval–when a given rotation is requested, the template is retrieved. Since the underlying structure no longer needs to be computed, the precise positions of the faces are updated according to the actual rotation angle. However, there are a potentially large number of θ values for which the structure changes, and the criticality map can become quite large for even relatively simple-seeming inputs. Further, expanding the rotation to Euler angles about two or three axes dramatically increases the space complexity of algorithm to the point of being impractical. As a result, while the algorithm is suitable for dealing with rotation about an arbitrary axis, once that axis is chosen, it is fixed unless one is willing to recompute the entire criticality map.

Lien proposes to update the facets of the Minkowski sum of convex polyhedra by identifying a linear subset of errors introduced through rotation; locating the remaining errors by forming connected components using the elements of the linear subset as seeds. The published work was a video abstract, as a preliminary to our later work [10].

Chapter 3: Efficient 2D Minkowski sums using the reduced convolution

3.1 Introduction

We begin the discussion of our work with [12] a convolution-based method that forms the foundation for our work in two-dimensions.

For computing the Minkowski sum ($\mathcal{M}$-sum) of non-convex shapes, many methods are based on the idea of convex decomposition. In these methods, the input models are decomposed into components. Because computing the $\mathcal{M}$-sum of convex shapes is easier than non-convex shapes, convex decomposition is widely used. The next step in this framework computes the pairwise $\mathcal{M}$-sums of the components. Finally, all these pairwise $\mathcal{M}$-sums are united to form the final $\mathcal{M}$-sum. Although conceptually simple, this method is usually not practical due to the size of the decomposition and the difficulty in implementing a robust union operation.

Convolution-based methods do not have these problems. The convolution of two shapes P and Q, denoted as $P \otimes Q$, is a set of line segments in 2D or facets in 3D that is generated by "combining" the segments or the facets of P and Q [30]. One can think of the convolution as the $\mathcal{M}$-sum that involves only the boundary, i.e., $P \otimes Q = \partial P \oplus \partial Q$. It is known that the convolution forms a superset of the $\mathcal{M}$-sum [28], i.e., $\partial(P \oplus Q) \subset P \otimes Q$. To obtain the $\mathcal{M}$-sum boundary, it is necessary to trim the line segments or the facets of the convolution.

In [12], we propose a new method for computing the 2D $\mathcal{M}$-sum of non-convex polygons. Our method can be viewed as a convolution-based approach. The main idea is to use the *reduced convolution* (defined later in Section 3.3) and extract the boundaries by using the topological properties of the $\mathcal{M}$-sum.

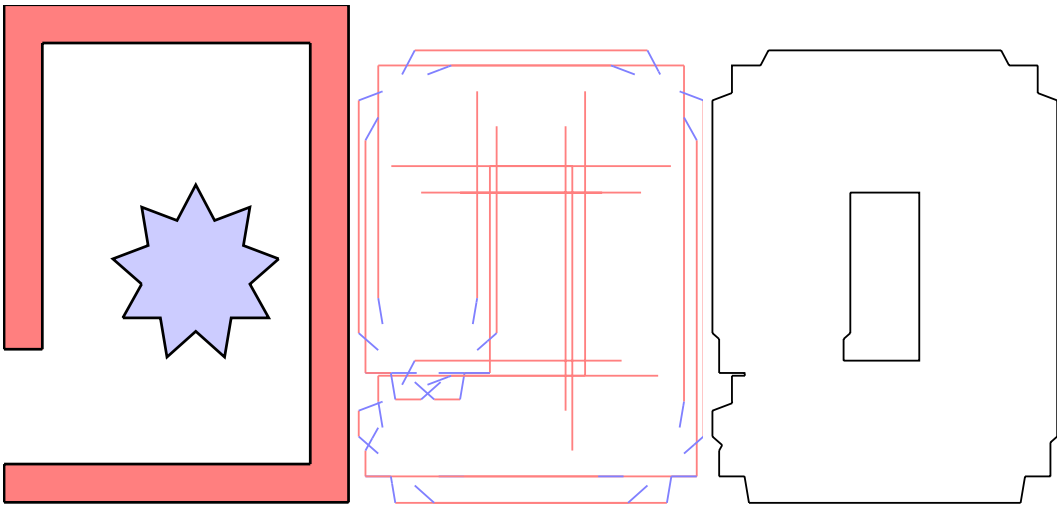


Figure 3.1: (a) Input polygons: star and fence. (b) The reduced convolution of star and fence. (c) The Minkowski sum of star and fence generated from the reduced convolution.

The main idea of the reduced convolution is inspired by the fact that, in most cases, the complexity of the complete convolution is much higher than the complexity of the final $\mathcal{M}$-sum boundary. This means that a large portion of the computation is spent on computing the arrangement induced by the complete convolution, and many elements in this arrangement are later on thrown away.

To improve the efficiency, an obvious approach is to avoid computing the complete convolution and its arrangement. However, the difficulty now becomes whether we can define a smaller set of the convolution while still being able to extract the $\mathcal{M}$-sum from this set.

Our method is designed to specifically avoid this waste of computation and address these difficulties. A detailed description of the proposed method can be found in Section 3.3. Finally, in Section 3.4, we experimentally demonstrate that the proposed method is more efficient than the existing methods.

Although this method is not dynamic, it forms the basis for some of our later work in dynamic Minkowski sums. It has also found direct use in applications. Elkeran [24] used our

Minkowski sum method in the process of solving the sheet nesting problem, and Guerrero, et al. [29], used it to implement a method for edit propagation across similar polygons in computer design work.

3.2 Notation

Let P and Q be simple polygons composed of n and m (counterclockwise) ordered vertices, respectively. We denote the vertices of P as $\{p_i\}$ and the edge that starts at vertex p_i as $e_i = \overline{p_i p_{i+1}}$. The edge e_i has two associated vectors, the vector from p_i to p_{i+1}, i.e., $\vec{v_i} = \overrightarrow{p_i\, p_{i+1}}$, and the outward normal $\vec{n_i}$. The definition for the vertices $\{q_j\}$ and edges of Q is the same.

3.3 Method

We propose a new method to compute the $\mathcal{M}$-sum of simple non-convex polygons. Similar to Wein's method [50], our method can be considered as a type of convolution-based approach. However, unlike Wein [50], the proposed method avoids computing (1) the complete convolution, (2) the arrangement of the segments of the convolution, and (3) the winding number for each arrangement cell.

Algorithm 3.3.1: $\mathcal{M}$-SUM(P, Q)

$R =$ Compute the reduced convolution of P and Q
$L =$ Extract orientable loops from R
$M =$ Filter boundaries from L
return (M)

Our method is sketched in Algorithm 3.3.1. Algorithm 3.3.1 first computes a subset of the segments that is from the convolution of the inputs. We call this subset a "reduced

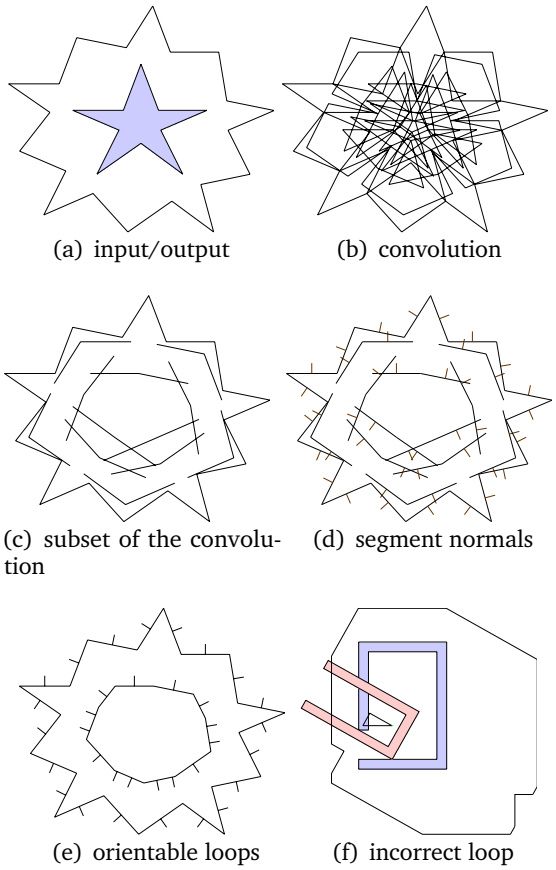


Figure 3.2: Steps for computing the $\mathcal{M}$-sum of two simple polygons. In (a), the boundary of the $\mathcal{M}$-sum of a star and a slightly rotated copy of itself is shown.

convolution."

Definition 3. *A* **reduced convolution** *is a set of segments* $\overline{p_i p_{i+1}} \oplus q_j$ *and* $p_k \oplus \overline{q_l q_{l+1}}$ *and* q_j *and* p_k *must be convex.*

Then Algorithm 3.3.1 identifies closed loops that are (1) non-overlapping and (2) *orientable*. These loops form potential boundaries of the $\mathcal{M}$-sum and are further filtered by analyzing their nesting relationship. Finally, the remaining boundaries are filtered by checking the intersections between the input polygons placed at the configurations along these loops. Fig. 3.2 illustrates these steps.

3.3.1 Reduced convolution

In the first step of the algorithm, we compute a subset of the segments of the convolution based on the following simple observation.

Observation 1. *Given a convolution segment $s = e_i \oplus q$ of an edge $e_i \in P$ and a vertex $q \in Q$, if q is a reflex vertex, s must not be a boundary of the $\mathcal{M}$-sum of P and Q. This observation remains true if $s = p \oplus e_j$, where $p \in P$ is a reflex vertex and $e_j \in Q$ is an edge.*

Proof. Let $\mathcal{S}$ be a set of segments formed by the end points of e_i and the edges incident to q. Because s must be incident to the segments $\mathcal{S}$, the vertex incident to both s and $\mathcal{S}$ is locally non-manifold. Moreover, by definition of convolution, s must be enclosed by the turning range of $\mathcal{S}$. Therefore, s cannot be on the boundary of the $\mathcal{M}$-sum. □

Figs. 3.2(b) and 3.2(c) show an example of the difference between the complete convolution and the reduced convolution. Because of the definition of a reflex angle, the number of edges that are compatible with any convex vertex in Q form a lower bound on the number of edges compatible with any reflex vertex in Q. Due to this, the number of segments filtered by Observation 1 is significant, and the size of the problem that we have to consider later is greatly reduced, in particular when the number of the reflex vertices is large. See the more detailed analysis later in this section.

3.3.2 Orientable Loop Extraction

Now, since the segments that we will be working with are no longer a complete convolution, we cannot apply the idea of computing the winding number for each arrangement cell to extract the $\mathcal{M}$-sum boundary as done in [50]. Instead, we proceed by defining two filters.

Observation 2. *We observe that the boundary of the Minkowski sum must be an orientable loop (if it encloses an area, either positive or negative).*

We say that a loop is orientable if all the normal directions of the edges in the loop are all either pointing inward or outward. Note that the segments we considered are edges

from P and Q, therefore, they are directional (as vertices in P and Q are ordered) and include normal directions pointing outward (to P or Q). Fig. 3.2(d) shows the normals of the segments. Therefore, given two adjacent segments $s = \{u, v\}$ and $s' = \{v, u'\}$ sharing an end point v, we can check whether s and s' belong to an orientable loop if

$$\overrightarrow{u\,v} \times \vec{n_s} = \overrightarrow{v\,u'} \times \vec{n_{s'}} \ , \tag{3.1}$$

where $\vec{n_x}$ is the normal vector of segment x, and $\times$ is the cross product. If s and s' satisfy Eq. 3.1, we say they are compatible segments.

To extract all orientable loops, we compute the intersections of the segments and split all segments at the intersections. A loop is then traced by starting at an arbitrary segment s that has not been considered and then iteratively including compatible segments adjacent to s. Note that there can be multiple compatible segments adjacent to s and all are incident to a single point v. This problem is in fact easy to handle since all $\mathcal{M}$-sum boundaries must be manifold. Thus, we simply pick the segment that makes the largest clockwise turn from s among all the incident segments. Fig. 3.2(e) shows the loops generated by this step.

Observation 3. *The loops must obey the nesting property, i.e., the loops that are* directly *enclosed by the external loop must be holes and will have negative areas, and the loops that are directly enclosed by the holes must have positive areas.*

This is because all loops we generated are non-overlapping (i.e., they don't intersect or touch) due to the manifold properties. The nesting property can be determined efficiently using a plane sweep algorithm, e.g., [4], in $O(n \log n)$ time for n segments. This filter removes the inner loop in Fig. 3.2(e) because it has a positive area.

3.3.3 Boundary Filtering

So far, we have introduced three quite efficient filters based on Observations 1 through 3. Unfortunately, not all of the remaining loops are boundaries of the $\mathcal{M}$-sum. For example,

the hole in Fig. 3.2(f) is a false loop. Therefore, we will have to resort to collision detection to remove all the false loops. That is, we will use the close relationship between the $\mathcal{M}$-sum boundary and the concept of "contact space" in robotics. Every point in the contact space represents a configuration that places the robot in contact with (but without colliding with) the obstacles. Given a translational robot P and obstacles Q, the contact space of P and Q can be represented as $\partial((-P) \oplus Q)$, where $-P = \{-p \mid p \in P\}$. In other words, if a point x is on the boundary of the $\mathcal{M}$-sum of two polygons P and Q, then the following condition must be true:

$$(-P^{\circ} + x) \cap Q^{\circ} = \emptyset \ ,$$

where Q° is the open set of Q (i.e., the interior, $Q - \partial Q$,) and $(P + x)$ denotes translating P to x. Fig. 3.2(f) shows a hole loop that passes all the filters except the last filter.

Although there are many methods to optimize the computation time for collision detection, collision detection is more time consuming than the previous filters. Fortunately, it is easy to show that only a single collision detection is needed to reject or accept a loop based on the following lemma.

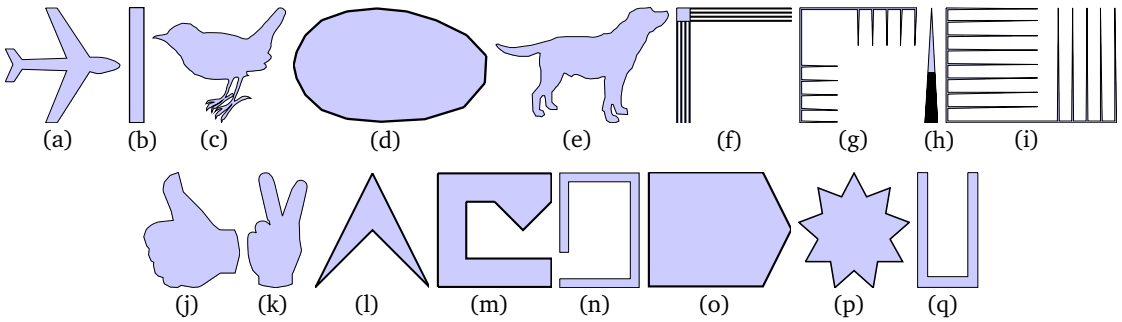
Lemma 1. *All the points on a false hole loop must make P collide with Q.*

Proof. Each loop must belong to a cell from the arrangement of the segments in the complete convolution. Moreover, all vertices in an arrangement cell must have the same winding number according to [50]. Therefore, a single point from each loop is sufficient to test if the loop is a true boundary or not. □

3.3.4 Complexity Analysis

When P and Q have n and m vertices which include n' and m' reflex vertices, respectively, there will be $2mn$ segments in the complete convolution; in the reduced convolution there are at most $(m - m')n + (n - n')m$ segments. That is, the arrangement of the reduced

convolution is at least 4 times less complex than that of the complete convolution when $n' = 1/2n$ and $m' = 1/2m$. Note that this analysis is based on the assumption that a convex vertex is compatible with $\Theta(n)$ edges and in the worst case that each segment will intersect all the other segments. In the examples that we will use in the experiment (shown in Fig. 3.3), the difference between the reduced and complete convolutions is more significant (e.g., "star/star" and "dog/bird"). The time complexity for computing the $\mathcal{M}$-sum of P and Q is $O\left((mn + I)\log(mn + I) + \ell T_{cd}\right)$, where $I = O(m^2n^2)$ is the complexity of the arrangement of the reduced convolution, ℓ is the number of loops, and $T_{cd} = O(mn)$ is the collision detection time in our implementation.



(a) airplane size=51	(b) bar 4	(c) bird 275	(d) blob 18	(e) dog 145	(f) G1 30
(g) G2 size=34	(h) G3 24	(i) G4 43	(j) hand1 57	(k) hand2 84	(l) insignia 4
(m) sewing size=10	(n) fence 10	(o) pentagon 5	(p) star 44	(q) U 8	

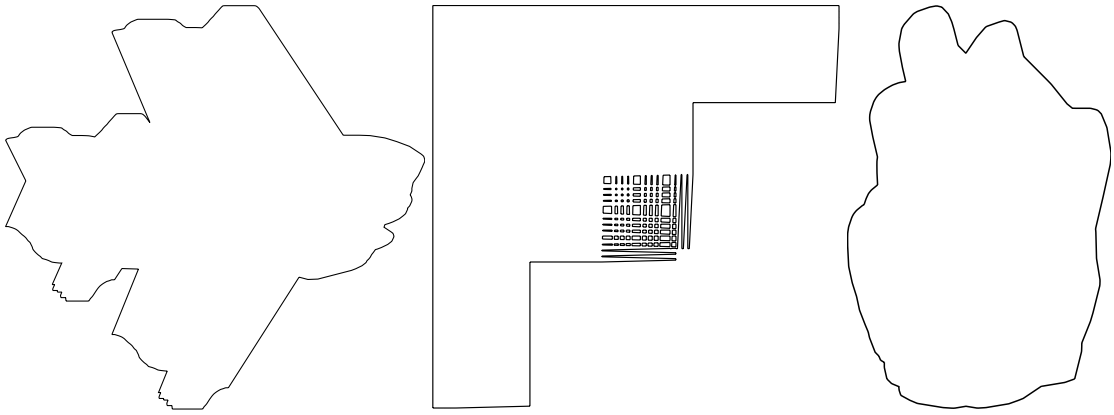
Figure 3.3: Models used in the experiments. The table shows the names and the sizes of the polygons.

3.4 Experimental results

In this section, we show that the computation time of the proposed method is more efficient than the traditional approach. All the experiments are performed on a machine with Intel

CPUs at 2.13 GHz with 4 GB RAM. Our implementation is coded in C++.

In our current implementation, the line segment intersection and the collision detection between P and Q are performed by exhaustively checking all pairs of segments. Figure 3.3 shows 17 models that we will use in the experiments. In Figure 3.4, we show three examples generated by the proposed method.



(a) bird⊕airplane (b) G1⊕G2 (c) hand1⊕hand2

Figure 3.4: Examples of the $\mathcal{M}$-sums generated by our program using the models in Fig. 3.3. (a) There are 1339 vertices and an external boundary. (b) There are 1204 vertices and 101 boundaries. (c) There are 375 vertices and an external boundary.

3.4.1 Computation Time

We first compare the computation time of the proposed method to the convolution implementation provided by CGAL [25]. As stated in the CGAL documentation, the CGAL code implements Wein's idea [50]. We use all the models in Fig. 3.3 in this experiment and compute the $\mathcal{M}$-sums from all pairs. The results are shown in Fig. 3.5.

In all examples, our method is always faster than CGAL. The smallest speedup is 1.3 from bar⊕bar, and the largest speedup is 42.4 from bird⊕hand1. In fact, from the results of this experiment, we observe that when the input models are larger (e.g., bird), our method

tends to gain more speedup. On average, our method takes 20.26 milliseconds to compute each $\mathcal{M}$-sum while CGAL takes 260.73 milliseconds.

In each figure we call out a few notable examples:

(A) $G3 \oplus monkey$ [smallest speedup]

(B) $monkey \oplus monkey$ [most complex convolution]

(C) $G1 \oplus G2$ [largest percent computation time performing collision detection]

(D) $hand1 \oplus hand2$ [least complex convolution]

(E) $bird1 \oplus hand1$ [largest speedup].

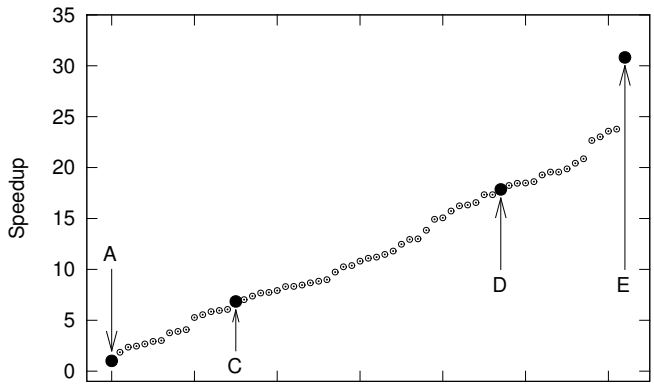


Figure 3.5: Speedup over CGAL implementation. Each dot is computed as t_{CGAL}/t_{ours}, where t_{CGAL} and t_{ours} are the computation times for a specific pair of models using CGAL and our method, respectively. Input pair (B) does not appear on this graph because CGAL failed to terminate on this pair.

Next, we look further into the behavior of the proposed method. We break down the computation time into three parts, i.e., (1) time for computing the intersections of the segments in the reduced convolution, (2) time for identifying the non-intersecting orientable loops, and (3) time for filtering hole boundaries. In this experiment, we use only the larger models in Fig. 3.3 (airplane, bird, dog, G1, G2, G3, G4, hand1, hand2, and star). The results

obtained from all pairs of these larger models are shown in Fig. 3.6.

One may be concerned that the use of collision detection for filtering out the remaining holes will be the bottleneck of the entire computation. Fig. 3.6 shows that this is in fact not the case. In all of the examples we have tested, the filtering step only takes less than 40% of the time. The bottleneck is in fact in finding the line segment intersections of the convolution. Recall that in our implementation both collision detection and line segment intersection are calculated by checking line segments exhaustively between all pairs.

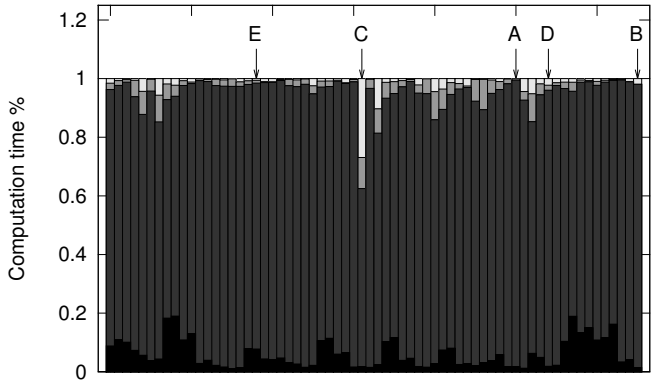


Figure 3.6: Normalized computation time for each step in our algorithm. The total computation for each $\mathcal{M}$-sum is normalized to 1. The dark grey part indicates the time for computing the intersections of the segments in the reduced convolution. The grey part is the time for identifying the loops. The light grey is the time for filtering hole boundaries. Only the larger models (airplane, bird, dog, G1, G2, G3, G4, hand1, hand2, and star) are used in this plot.

3.4.2 Reduced Convolution vs. Complete Convolution

In this experiment, we show that the size of the reduced convolution is the key that the proposed method is more efficient. Theoretically, the reduced convolution is at most half of the complete convolution. However, this analysis (in Section 3.3.4) is based on the assumption that a convex vertex is compatible with $\Theta(n)$ edges. Practically, this assumption

may be off. Fig. 3.7 shows exactly this. Again, we use larger models in Fig. 3.3 and compute the $\mathcal{M}$-sums of all pairs. We study the differences when the reduced convolution and the complete convolution are used.

Fig. 3.7 clearly shows that the size of the reduced convolution is significantly smaller than that of the regular convolution. Note that the y axis in this plot is in logarithmic scale. In the best case (bird$\oplus$dog), the reduced convolution is 13.13 times smaller. In this case, the reduced convolution has 2,921 segments and the convolution has 38,342 segments. In the worst case (G3$\oplus$G3), the reduced convolution is only 1.98 times smaller. In this case, the reduced convolution has 320 segments and the convolution has 632 segments.

Note that this experiment only studies the size of the convolutions. The discrepancy between the size of the arrangement of the reduced convolutions and that of the complete convolutions will be even larger.

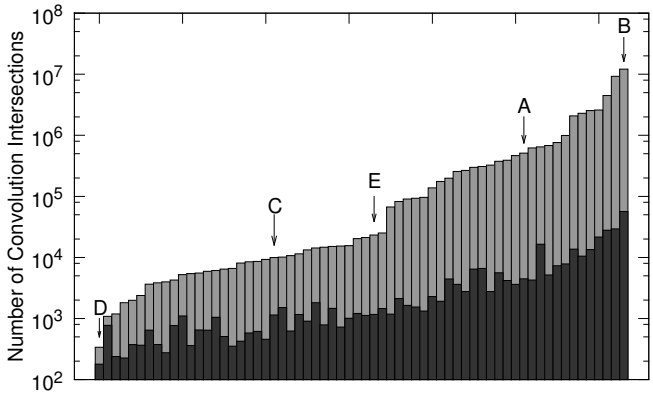


Figure 3.7: Number of segments in the reduced convolution and the complete convolution. The size of the reduced convolution is significantly smaller than that of the regular convolution. Note that the y axis is in logarithmic scale. Only the larger models (airplane, bird, dog, G1, G2, G3, G4, hand1, hand2, and star) are used in this plot.

Chapter 4: Computing the configuration space of rotating polygons

4.1 Introduction

The configuration space ($\mathcal{C}$-space) of a movable object P is the enumeration of all configurations of P. A closed subset of the $\mathcal{C}$-space that causes P to collide with obstacles Q is called $\mathcal{C}$-space obstacle ($\mathcal{C}$-obst). It is the mapping from the workspace obstacles to the $\mathcal{C}$-obst that has interested researchers since the late 1970s.

It is well known that computing an explicit geometric representation of $\mathcal{C}$-space is intractable for objects with high degree of freedom [18], and researchers have been successfully solving difficult problems without computing the $\mathcal{C}$-obst, e.g., using probabilistic motion planners (see [34]). However, an explicit representation of $\mathcal{C}$-space remains important to many problems, including problems that require complete motion planners (e.g., assembly/disassembly), CAD (e.g., Caine's design of shape [16]), virtual prototyping, object placement [3] and containment [5]. In addition, $\mathcal{C}$-space mapping is fundamental to basic geometric operations, such as continuous collision detection and generalized penetration depth estimation. Since the early 1980s to the mid-1990s, many researchers have proposed several methods to compute and approximate various types of representation of the $\mathcal{C}$-obst. However, not until more recently have newer developments (e.g. the idea of *configuration products* by Nelaturi and Shapiro [46]) been made toward improving and generalizing these methods, partly because the need for an explicit representation of $\mathcal{C}$-obst diminished after the development of sampling-based motion planners. See the survey by Wise and Bowyer [51] for a complete review on these earlier works and see Section 2 for a brief overview of the related and recent works.

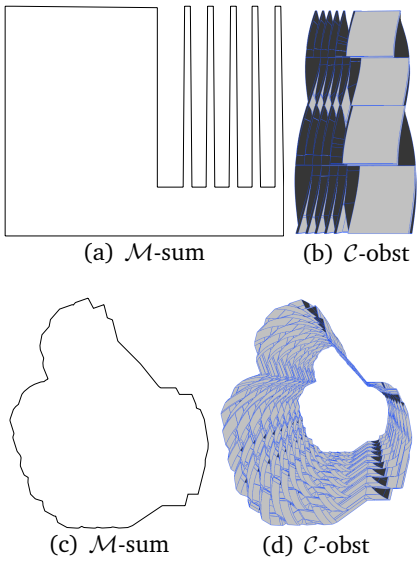


Figure 4.1: Examples. (a) and (b) are generated from bar and grate 3 in Fig. 4.3. (c) and (d) are generated from star and hand in Fig. 4.3.

There exist few methods that focus on the exact representations of $\mathcal{C}$-obst of polygons, e.g., [3, 15, 21, 31]. In these techniques, the methods proposed by Avnaim et al. [3] and Brost [15] are the ones closely related to our work. Avnaim et al. [3] proposed to compute $\partial\mathcal{C}$-obst using contact regions. A contact region is computed between a vertex of P and an edge of Q or vice versa. Though $\mathcal{M}$-sums are not used in this method, a similar configuration space is produced. Their algorithm has time complexity $O(n^3m^3 \log nm)$ for polygons with n and m vertices. Similar to Avnaim et al. [3], Brost [15] also considered all possible contacts. In both methods, a contact region can have zero area on $\mathcal{C}$-obst. This means that the entire contact region is trimmed. In fact, even for simple shapes (for example the star shape shown in Fig. 3.2), many contact regions will not be on the surface of $\mathcal{C}$-obst. As a result, significant computation is wasted on finding the intersections between contact regions (which is a computational expensive operation).

In [9] we present a new method for mapping 2-d polygons to their 3-d $\mathcal{C}$-obst. Our method represents the boundary ($\partial\mathcal{C}$-obst) of $\mathcal{C}$-obst as a set of *ruled surfaces*. The proposed method is simpler to implement than the existing methods in the literature [3, 15] and is

often more efficient. These main advantages are provided by a new algorithm that allows us to extract the Minkowski sum ($\mathcal{M}$-sum) boundary from the *reduced convolution* (defined in Section 4.3.2) of the input polygons. As a warm-up, we will first show that the $\mathcal{C}$-obst of convex polygons can be computed using the idea of convolution at *critical orientations* (in Section 4.3.1). We then show that the $\mathcal{M}$-sum of two simple non-convex polygons can be computed efficiently using the filtering-based approach from the reduced convolution. Finally, the $\partial\mathcal{C}$-obst of non-convex polygons is constructed by updating the $\mathcal{M}$-sum at the critical orientations. The time complexity of our method is $O(m^3n^3 + bT_{cd})$ for polygons with m and n vertices, where b is the number of boundaries of $\mathcal{C}$-obst, and T_{cd} is the time for a single collision query. We also show that the resulting $\mathcal{C}$-obst can be used for efficiently estimating the generalized penetration depth by computing the closest feature between the query point and the ruled surfaces (Section 4.4.2).

4.2 Preliminaries

We assume that P is movable while Q is stationary. Both P and Q are simple polygons composed of n and m (counterclockwise) ordered vertices, respectively. Our approach is based on computing and updating the $\mathcal{M}$-sums using the reduced convolution approach described in Chapter 3.

Without loss of generality, we assume that P rotates counterclockwise about c, and c is the world origin, so that $c_x = c_y = 0$.

4.3 Our Methods

We will first discuss the case of convex polygons in Section 4.3.1 and then extend the ideas the handle non-convex polygons in Section 4.3.2.

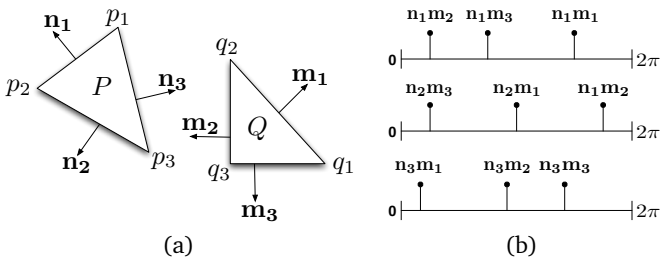


Figure 4.2: (a) Two convex polygons P and Q shown with the edges outward normals. (b) Events for $\vec{n_1}$, $\vec{n_2}$, and $\vec{n_3}$ (from top to bottom) when P rotates counterclockwise from 0 to 2π. For example, when P rotates $\pi/4$, $\vec{n_1}$ and $\vec{m_2}$ (and $\vec{n_2}$ and $\vec{m_3}$) become aligned and two events are issued.

4.3.1 $\mathcal{C}$-obst of Convex Polygons

Given two convex polygons P and Q (see Fig. 4.2(a)), an edge of their convolution is the sum of an edge $\overline{p_i p_{i+1}}$ of P and a vertex q_j of Q or vice versa. We let θ_0 be the orientation when an edge/vertex pair is born until it dies at θ_1 when P rotates counterclockwise, and each edge/vertex pair forms a *parameterizable ruled surface* (i.e. contact region). Let $p_i = (x_0, y_0)$ and $p_{i+1} = (x_1, y_1)$. We take a vector $\vec{v} = \overrightarrow{p_i\, p_{i+1}}$ and a vector $\vec{t} = \overrightarrow{O\, q_j}$, where O is the world origin. Then the surface defined by the pair ($\overline{p_i p_{i+1}}$ and q_j) is parameterized as:

$$S_R(r, \theta) = \begin{bmatrix} (x_0 + rv_x)\cos\theta - (y_0 + rv_y)\sin\theta + t_x \\ (x_0 + rv_x)\sin\theta + (y_0 + rv_y)\cos\theta + t_y \\ \theta \end{bmatrix}, \tag{4.1}$$

where $r \in [0, 1]$, $\theta \in [\theta_0, \theta_1)$.

Surfaces are also formed by the edges of Q as P rotates. Similarly, we let $q_j = (x_0, y_0)$ and $q_{j+1} = (x_1, y_1)$. We take the vector $\vec{v} = \overrightarrow{q_j\, q_{j+1}}$ and a vector $\vec{t} = \overrightarrow{O\, p_i}$. The surface for

the pair p_i and $\overline{q_j q_{j+1}}$ is parameterized as:

$$S_N(r,\theta) = \begin{bmatrix} y_0 \cos\theta - x_0 \sin\theta + t_x + r v_x \\ x_0 \cos\theta + y_0 \sin\theta + t_y + r v_y \\ \theta \end{bmatrix}. \tag{4.2}$$

In order to support operations like distance query and line intersection, each surface is stored as a tuple $(p, q, \theta_0, \theta_1)$, where p and q are the indices to the vertices and edges of P and Q, and θ_0 and θ_1 define the *birth and death* orientations of the surface. To construct the surfaces in this representation, we use a sweeping algorithm that updates the convolution at critical orientations (events). Fig. 4.2(b) shows all the events for each edge of P. To handle each event, we delete two segments from the convolution and create two new segments. For example, at event $n_3 \vec{m}_1$ in Fig. 4.2(b), the pairs $\langle \overline{p_3 p_1}, q_1 \rangle$ and $\langle p_1, \overline{q_1 q_2} \rangle$ both die and the pairs $\langle \overline{p_3 p_1}, q_2 \rangle$ and $\langle p_3, \overline{q_1 q_2} \rangle$ are both born. Note that these changes are local, therefore each event can be handled in a constant time, and there can be at most mn events. Moreover the events for each edge of P is simply an offset copy of the normals of Q, so all (sorted) events can be built in linear time. Therefore, the entire computation takes only $\Theta(nm)$ time.

For two convex polygons, the surfaces trivially form $\partial\mathcal{C}$-obst, as the surfaces will never penetrate into the interior of the $\mathcal{C}$-obst. However, in the case where one or both of the inputs are non-convex, this is not guaranteed to be the case. This poses fundamental problems in computing the penetration depth on such a solid; for example, the closest point on a non-manifold hull to a query point inside the solid may consequently still be on the interior of the solid.

4.3.2 $\mathcal{C}$-obst of General Simple Polygons

We have already discussed how to compute the $\mathcal{M}$-sum for polygons without rotation in Chapter 3. We now generalize generalize the approach to consider rotation.

Handling Rotation

Similar to the algorithm that we proposed for computing the $\partial\mathcal{C}$-obst for convex shapes, the algorithm for the $\partial\mathcal{C}$-obst for non-convex shapes also consists of $\Theta(mn)$ events for creating and deleting each contact patch. Each patch is generated by a segment (with varying length) in the reduced convolution. In addition to these events, the intersection of line segments (from the reduced convolution) also changes during the rotation of P, and these changes can affect the topological structure of the $\mathcal{M}$-sum. Therefore, the second type of event for a given segment s is a list of rotations $\{\theta_i\}$ where the intersection status of s changes (e.g., s starts to intersect or stops intersecting with a segment) when P rotates from 0 to 2π. There can be $O(C^2)$ such events, where C is the size of the reduced convolution.

The data structure that we use for representing the surface is also a tuple $(p, q, \theta_0, \theta_1, s_1, s_2)$, where p, q, θ_0, and θ_1 are the same as the convex case, and s_1 and s_2 are indices to the convolution segments intersecting with the segment between θ_0, and θ_1. We use the same sweeping algorithm to construct this data structure. To handle the events where a segment s is created (or deleted), we simply add (or remove) s to the reduced convolution and add (or remove) the intersections due to s. To handle the second type of event, intersections due to the events are updated.

When a new intersection occurs, s is split into two segments at the intersection point. One of the new segments resulting from this split may be degenerate in the case of a new intersection occurring at an endpoint of s, however this does not require any special case handling. The degenerate segment will expand with the movement of the intersecting segment into a proper line segment and generate the correct ruled surface. An additional benefit to this is that the result of the splitting operations guarantees us that at most two line segments intersect each of the resulting segments of the convolution. This ensures that our data structure is always sufficient to represent a particular surface patch of the $\mathcal{C}$-obst.

For both events, we check if the event site (i.e., the new or dead intersection) is locally orientable and manifold to decide if a loop (of constant size) should be created or deleted as

described above. Note that we will skip the last two filters (i.e., the polygon nesting and the collision detection filters) during sweeping. Both filters will only be needed at the end of the sweep to reject the false 3-d hole boundaries. Similarly, we can show that only one (2-d) point is needed to verify each 3-d boundary; the proof is similar to Lemma 1.

An example of the results generated by our method is shown in Fig. 4.4. From the figures, we can see that the interior part of the $\mathcal{C}$-obst is hollow and all the extra parts of the reduced convolution are correctly removed.

In the rest of this section, we will briefly discuss how to detect the second type of event. Each of these events can be found in constant time, though the computation requires us to classify the types of edges and surfaces, i.e., S_R and S_N in Eqs. 4.1 and 4.2, since they move (and rotate) in different ways.

Consider two rotating edges in the convolution e_1 and e_2 that may intersect at some θ. We let θ_0 be the first value of θ for which e_1 and e_2 are both alive. An edge e_i lies along a line L_i whose equation is $y_i(\theta) = m_i(\theta)x_i(\theta) + b_i(\theta)$. The intersection $(x(\theta), y(\theta))$ of L_i can be computed so that

$$x(\theta) = \frac{b_2(\theta_0) - b_1(\theta)}{m_1(\theta) - m_2(\theta_0)} \ ,$$

where $b_i(\theta) = x_0 \cos\theta + y_0 \sin\theta + t_y - m_i(\theta)(y_0 \cos\theta - x_0 \sin\theta + t_x)$ and $m_i(\theta) = \frac{1+m_i \tan\theta}{m_i - \tan\theta}$, and m_i is the initial slope of L_i. It is trivial to compute $y(\theta)$ from $x(\theta)$. Then from the intersection of the lines, we solve for θ such that the intersection $(x(\theta), y(\theta))$ will fall into the range of the line segments e_1 and e_2. This is done by classifying the segments into three cases that involving *rotating* and *non-rotating* edges. We say that the edges that create S_R surfaces are rotating edges and the edges that create S_N surfaces are non-rotating edges. Therefore e_1 and e_2 can be either (1) both rotating edges, (2) both non-rotating edges or (3) a rotating and non-rotating pair. In certain cases, the segments can be checked quickly to determine if they ever intersect. The details are shown in the Appendix.

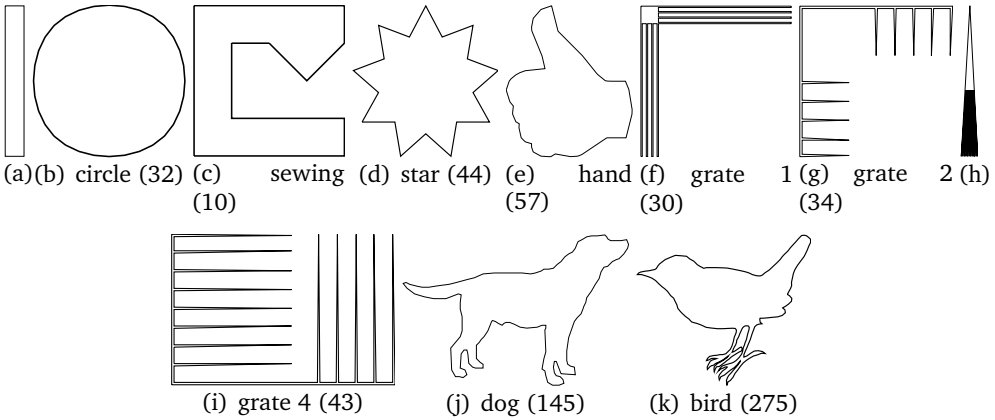


Figure 4.3: Models used in the experiments, a subset of those used in Chapter 3. (a) bar (4); (h) grate 3 (34). The numbers in the parentheses are the size of the polygons. Some of these models are inspired by those in [32, 50].

4.4 Experimental Results, Application and Discussion

4.4.1 Results

We have implemented the proposed method in C++. In this section, we reproduce from [9] results that we obtained from this implementation using the examples shown in Fig. 4.3. In these examples, there are two convex polygons and 9 non-convex polygons. The number of the vertices of each polygon is also shown. Some of these models are inspired by those in [32, 50]. All the experiments are performed on a PC with Intel CPUs at 2.13 GHz with 4 GB RAM.

In Table 4.1, we show the computation time for constructing $\partial\mathcal{C}$-obst using the proposed method. The running times range from a fraction of a second to close to an hour. Since we have no other implementation to compare to and our implementation is highly unoptimized (for example, our collision detection takes $\Theta(mn)$ for each collision check), it is important to look at these running times relatively. Therefore, we list the number of ruled surfaces before trimming (N_s), the number of ruled surfaces on the final $\partial\mathcal{C}$-obst (n_s), and the number (external and hole) of $\mathcal{C}$-obst boundaries (n_b). From the values of N_s, n_s, and n_b, it is clear all of them can affect the computation time. For example, both "star/star" and "grate 1/grate

Table 4.1: Experimental results for $\mathcal{C}$-space mapping. Here, N_s is the number of ruled surfaces before trimming, n_s is the number of ruled surfaces on $\partial\mathcal{C}$-obst, n_b is the number (both external and hole) of $\mathcal{C}$-obst boundaries, and t is the total computation time in seconds.

P/Q	bar/circle	bar/sewing	star/star	star/hand	grate 1/grate 2	bar/grate 4
t	0.1	0.05	4.9	6.8	4.9	0.3
N_s	256	68	2288	3286	1028	244
n_s	256	82	3499	3034	4097	1027
n_b	1	1	1	1	126	17

P/Q	grate 3/grate 4	dog/bird
t	21.7	3350.6
N_s	991	39145
n_s	1947	18500
n_b	39	1

2" take about the same time to compute, but the number of ruled surface patches in "grate 1/grate 2" is half of that in "star/star." Therefore, it is the large n_b in "grate 1/grate 2" that increases the computation time. Moreover, it is clear that the reason that the "dog/bird" takes nearly an hour to finish is because of N_s, which is about 40 times the N_s of "grate 1/grate 2" and "grate 3/grate 4." One single example that we cannot explain from Table 4.1 is the time difference between "grate 1/grate 2" and "grate 3/grate 4." Both N_s and n_s are smaller and n_b is larger in "grate 1/grate 2."

Fortunately, we can explain this in Table 4.2. In Table 4.2, we show the number of segments and the number of intersections in both complete convolution ($N_\otimes$ and $I_\otimes$, resp.) and reduced convolution ($n_\otimes$ and $i_\otimes$, resp.) at the orientation shown in Fig. 4.3. The reason that "grate 1/grate 2" takes less time to compute than "grate 3/grate 4" does is because "grate 1/grate 2" tends to have smaller $i_\otimes$.

An important observation from Table 4.2 is the significant difference between $N_\otimes$ and $n_\otimes$. As we have mentioned above, when the full convolution is used, a large number of ruled surfaces will be generated and many of these ruled surfaces are not on $\partial\mathcal{C}$-obst. As a result, much computation is wasted on computing the intersections between these surfaces. To make the problem worse, the values for $I_\otimes$ and $i_\otimes$ show that these unnecessary surfaces

produce drastically more intersections than those left in the reduced convolution. This difference can also be observed in Fig. 3.2. This property distinguishes our method from the existing methods [3, 15], which consider all contact regions (surfaces). Therefore, we believe that our method is more efficient.

Table 4.2: Experimental results for $\mathcal{M}$-sum computation. Here, $N_{\otimes}$ is the number of segments in the convolution, $n_{\otimes}$ is the number of segments in the reduced convolution, $I_{\otimes}$ is the number intersections in the convolution and $i_{\otimes}$ is the number intersections in the reduced convolution.

P/Q	bar/circle	bar/sewing	star/star	star/hand	grate 1/grate 2	bar/grate4
$N_{\otimes}$	36	42	1608	1689	1394	191
$I_{\otimes}$	36	33	1300	2758	5243	131
$n_{\otimes}$	36	30	382	281	469	92
$i_{\otimes}$	36	22	257	297	1204	77

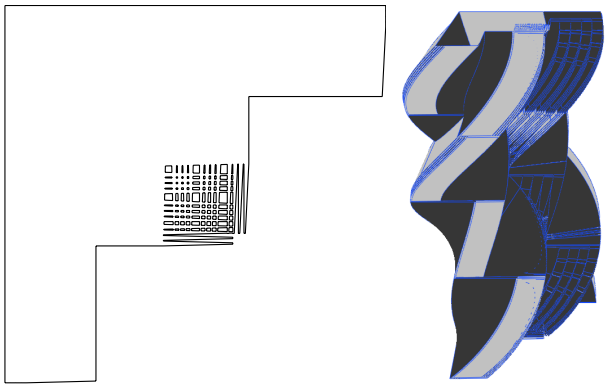
P/Q	grate 3/grate 4	dog/bird
$N_{\otimes}$	1162	38342
$I_{\otimes}$	10136	255635
$n_{\otimes}$	400	2921
$i_{\otimes}$	1544	3742

4.4.2 Application: Generalized Penetration Depth Estimation

The parameterizations in Eqs. 4.1 and 4.2 also yield distance functions in r and θ which can be used to find the minimum distance to a given facet relatively easily. Let p be a query point and let $f(r) = (x_0 + rv_x)$, $g(r) = (y_0 + rv_y)$, $F(r) = g(r)\cos\theta - f(r)\sin\theta$, and $G(r) = f(r)\cos\theta + g(r)\sin\theta$, then the square distance $d(r, \theta, p)$ for the rotating edges:

$$d(r, \theta, p) = (F(r) + t_x - p_x)^2 + (G(r) + t_y - p_y)^2 + w^2(\theta - p_z)^2$$

If we fix r, then d is a very well-behaved sinusoid, and while there does not seem to be a closed-form solution for the global minimum, it is easy to find the minimum using simple



(a) $\mathcal{M}$-sum (b) $\mathcal{C}$-obst

Figure 4.4: The $\mathcal{M}$-sum and $\mathcal{C}$-obst of grate 1 and grate 2 in Fig. 4.3. The darker (lighter) patches in (b) are S_R (S_N) surfaces.

gradient descent. If by contrast we fix θ, then d is simply quadratic in r, and finding the global minimum on $[0, 1]$ is also quite easy.

Computing $d(r, \theta)$. In the case of S_R (see Fig. 4.5(a)), the regularity of the surface of the distance function allows us to easily calculate a global minimum by finding θ values for the global minimums at $r = 0$ and $r = 1$ by gradient descent, then finding the global minimums for r when we fix θ at the values found by fixing r initially. Picking the minimum of the yielded values gives us the global minimum of the distance function, as well as yielding r and θ values which explicitly give us the closest point on the facet (see Fig. 4.5(c)). The distance function follows similarly for S_N (see Fig. 4.5(b)), except that because the r term is independent of the rotation, the surface is somewhat more regular. We still end up with no clear closed-form solution for the sinusoid however, so we must solve for the minimum using gradient descent as above.

In the case of non-convex polygons, a surface may have a left-r-bound function $r_{min}(\theta)$ and a right-r-bound function $r_{max}(\theta)$ that describe how its non-manifold intersections move as θ changes, so that its associated facet is r-bounded at a given θ by $[\max\{0, r_{min}(\theta)\}, \min\{1, r_{max}(\theta)\}]$. These same r-bounds apply to the distance function. As a consequence, finding seed values for r and θ in the general case is more complicated. To deal with this issue, we choose

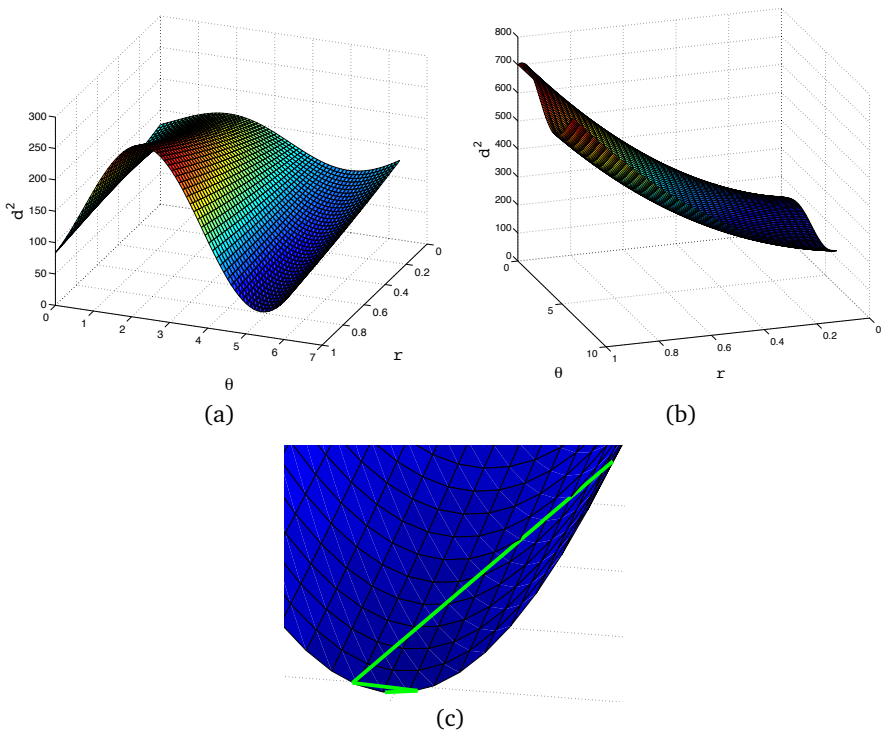


Figure 4.5: (a) An example distance function for S_R, $c = (0, 0), p = (10, -5, \pi)$, $v = (1, 4)$, $x_0 = 1, y_0 = 1$. (b) An example distance function for S_N, same parameters as (a) except that v is elongated. (c) An example gradient descent for the sinusoid portion zoomed in on the gradient descent. The descent converges in just 5 iterations (two iterations in which only step-size is adjusted), and in this case finds not only the local minimum for the sinusoid at r_{max}, but also the global minimum.

to seed at regular intervals. Let segment e have its birth at θ_0 and death at θ_1, then seed values are taken for $\theta \in \{k\frac{|(\theta_0-\theta_1)|}{8} : k \in \mathbb{Z}, 0 \leq k \leq 8\}$. For each of these θ values, we seed at $\max\{0, r_{min}(\theta)\}$, $\min\{1, r_{max}(\theta)\}$, and $(\max\{0, r_{min}(\theta)\} + \min\{1, r_{max}(\theta)\})/2 \pm \epsilon$, just to the left and right of the medial axis of the r-bounds.

This gives us a total of 36 seeds per surface. We use so many seeds largely because the r-bounds are irregular enough that some descents may get caught along the boundary. This spread however provides good coverage. Because of the regularity of the surface itself, a particular iteration of the gradient descent tends to converge in a small number of iterations and so the total cost of the gradient descent is relatively low in any case.

Table 4.3: Results for penetration depth estimation. Here ϵ is the avg. distance error, t is the avg. query time over 1000 queries, and T is the time to pre-compute the distances for all samples.

P/Q	bar/circle	bar/sewing	star/star	grate 1/grate 2
ϵ	0.000004	0.00067	0.0004	0.0001
t	7.8ms	6.5ms	8.5ms	12.0ms
T	50.1s	80.1s	398.2s	862.2s

Computing penetration depth. Given a configuration p of P, we would like to find the closest feature on $\partial\mathcal{C}$-obst. This problem can be decomposed into two steps: (1) find the closest surface f to p and (2) find the closest point on f to p. We have already proposed a method for the second step. For finding the closest surface, ideally, we can precompute the *Voronoi tessellation* of the space using each surface as a site, and then find which cell q is in. However, both computing the tessellation and finding the enclosing cell seem to be difficult. The only properties that we know are that the boundaries of the tessellation are also ruled surfaces, and each cell forms a single connected component. Based on these properties, we propose a sampling-based approach. Initially, a set of uniformly distributed samples are taken, and the closest surface for each sample point is computed offline using a brute-force search (through all surfaces). Each query point is then categorized by its k nearest neighbors, and only the $n \leq k$ surfaces associated with those neighbors are checked. For the results in Table 4.3, we set $k = 10$ experimentally. For convex polygons, this approach yields a very high rate (98.3% for bar/circle) of identifying the actual closest facet and low average error values ($< 10^{-5}$) when a facet other than the closest is chosen for distance comparison. For non-convex polygons, this approach still yields very high rate ($> 95.7\%$) of identifying the actual closest facet and low average error values ($< 10^{-3}$). The accuracy and error are estimated by comparing to the results of the brute force approach.

4.4.3 Complexity Analysis

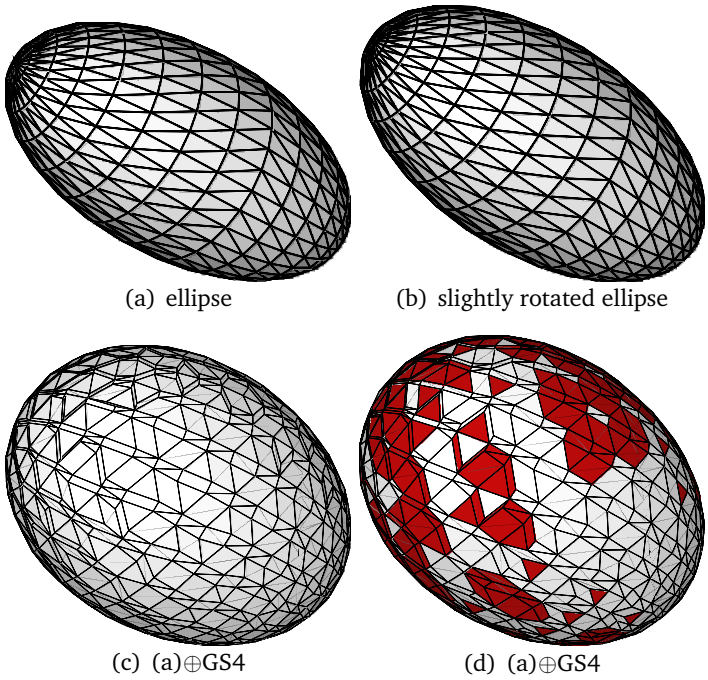
When P and Q have n and m vertices which include n' and m' reflex vertices, respectively, there will be $2mn$ segments in the complete convolution; in the reduced convolution there are at most $(m - m')n + (n - n')m$ segments. That is, the arrangement of the reduced convolution is at least 4 times less complex than that of the complete convolution when $n' = 1/2n$ and $m' = 1/2m$. Thus, the reduction will further reduce the complexity of the arrangement of 3-d rule-surface patches by at least 8 times. Note that this analysis is based on the assumption that a convex vertex is compatible with $\Theta(n)$ edges and in the worst case that each segment will intersect all the other segments. In the examples that we have above, the difference between the reduced and complete convolutions is more significant (e.g., "star/star" and "dog/bird"). The time complexity for computing the $\mathcal{M}$-sum of P and Q is $O\left((mn + I)\log(mn + I) + \ell T_{cd}\right)$, where $I = O(m^2n^2)$ is the complexity of the arrangement of the reduced convolution, ℓ is the number of loops, and $T_{cd} = O(mn)$ is the collision detection time in our implementation. The time complexity for computing the $\mathcal{C}$-obst of P and Q is $O\left((mn + I)\log(mn + I) + m^2n^2T_e + bT_{cd}\right)$, where b is the number of (hole) boundaries in the $\mathcal{C}$-obst, and $T_e = O(mn)$ is the time for handling each event (i.e., finding all new/dead intersections and update the $\mathcal{M}$-sum locally near the intersections).

Chapter 5: Dynamic rotation of convex objects

5.1 Introduction

We are also interested in a method that can efficiently compute the Minkowski sum of *rotating convex polyhedra*, as well as generalized 2D polygons. Computing the Minkowski sum of polyhedra undergoing rotations can be found in many problems, such as *general penetration depth* estimation [55] for physically-based simulation and *configuration-space obstacle* mapping [51] for robotic motion planning . Figure 5.1 shows an example of the Minkowski sums before and after rotating the ellipse.

The main challenge of computing the Minkowski sum of two rotating polyhedra comes from that fact that the Minkowski sum can be dramatically different after the input polyhedra rotate. Therefore, existing methods simply re-compute a new Minkowski sum every time P or Q rotates. For example, this approach is traditionally used to slice the $\mathcal{C}$-space obstacles ($\mathcal{C}$-obst) in motion planning. When the rotation of the robot is considered, $\mathcal{C}$-obst are approximated by repetitively computing the Minkowski sums of the robot with different orientations. These Minkowski sums are usually separated by a *fixed rotational resolution*. A main problem of re-computing the Minkowski sum from scratch is that it requires the same amount of computation even when a small amount of rotation is applied to P or Q.



(a) ellipse (b) slightly rotated ellipse

(c) (a)⊕GS4 (d) (a)⊕GS4

Figure 5.1: The Minkowski sums of a rotating ellipse and a sphere (GS4, shown in Fig. 5.3). The ellipse in (b) is rotated by $\pi/40$ from (a). The dark (red) facets in (d) are the differences between (c) and (d).

Our work is motivated by the observation described above. Thus, our objective is to compute the Minkowski sums of rotating convex polyhedra without re-computing the entire Minkowski sum repetitively. The main idea in our method is to generate the Minkowski sum from the existing Minkowski sum. More specifically, we generate the new Minkowski sum by correcting the "errors" introduced by rotation.

In theory, computing the Minkowski sum of two convex shapes P and Q will take $O(mn \log mn)$ time by overlaying the Gaussian maps of P and Q with complexities $O(m)$ and $O(n)$, respectively [26]. It is also known that the (space) complexity of the Minkowski sum of the same P and Q is $O(mn)$ [26]. Therefore, we expect an algorithm, similar to ours, that updates the Minkowski sum, instead of re-computing from scratch, will be $O(\log mn)$ faster than the traditional (brute force) approach.

Our Contribution. In [10], we demonstrate a method that provides the desired properties mentioned above. We call this method: DYMSUM (**dy**namic **M**inkowski **sum**). We show that DYMSUM is significantly more efficient than the naïve method of re-computing the Minkowski sum from scratch, in particular when the size of the input polyhedra are large and when the rotation angle is small between frames. From our experimental results, we show that the computation time of DYMSUM grows slowly (e.g., linearly if inputs are cubes) with respect to the size of the input comparing to the naïve approach (see Section 5.4). A preliminary version of this work can be found in a video abstract [39]. Although we focus on rotation of convex shapes, DYMSUM serves as the foundation for our work in scaling convex polyedra. We then demonstrate extensions of the work to non-convex shapes using a convex polyhedral mapping.

5.2 A Brute Force Method

Without loss of generality, we assume that P is movable while Q is stationary. We let P_s and P_t be two copies of P at two configurations s and t with distinct orientations. Our goal is to compute $M_t = P_t \oplus Q$ from $M_s = P_s \oplus Q$. Moreover, the computation time of the Minkowski sum should be sensitive to the orientation difference between P_s and P_t, i.e., the smaller the difference between P_s and P_t, the faster the computation of M_t.

Computing the Minkowski sum of two convex shapes is usually based on the idea of overlaying two Gaussian maps of the inputs. The Gaussian map $g(P)$ of a polyhedron P is a sub-division of $\mathbb{S}^2$. One can think $g(P)$ and P as dual to one another. That is, each face f of P with the outward normal n_f corresponds to a vertex $g(f) \in g(P)$ with coordinate n_f, and each vertex v of P corresponds to a face $g(v) \in g(P)$ bounded by the normals of the faces incident to v. When we overlay two Gaussian maps $g(P)$ and $g(Q)$, a vertex v in $g(P)$ must be associated with exactly one face in $g(Q)$ that encloses v and vice versa. Moreover, the edges in $g(P)$ and $g(Q)$ can also intersect.

The facets of a Minkowski sum are defined exactly by these two types of interactions

between $g(P)$ and $g(Q)$: the facets generated from a facet of P and a vertex of Q or vice versa, called fv-facets; and the facets generated from a pair of edges from P and Q, respectively, called ee-facets. A facet f and a vertex v produce an fv-facet if and only if the normal of f is a conical combination of the normals of the facets incident to v. Similarly, a pair of edges e_1 and e_2 form an ee-facet if and only if the cross product of vectors parallel to e_1 and e_2 is a convex combination of the normals of the facets incident to e_1 and e_2.

These criteria allow us to test if a given pair of features (a facet/vertex pair or an edge pair) will produce a Minkowski sum facet by checking only the neighborhood of these features. Given a pair of features (facet/vertex or edge/edge), we say that the features are *compatible* if they form either an fv-facet or an ee-facet. When P_s transforms to P_t, some facets (i.e., pairs of features) in M_s will no longer be compatible. We call these facets the "errors" introduced by rotation.

A brute force algorithm, which is used in all existing methods except [45], computes the Minkowski sums from P_s and P_t without considering the correspondences between them as shown in Algorithm 5.2.1. Given P_s and Q and the existing Minkowski sum M_s. Algorithm 5.2.1 rotates P_s by θ to obtain P_t. Then it uses an existing Minkowski sum algorithm to compute M_t.

Algorithm 5.2.1: BRUTEFORCE(M_s, P_s, Q, θ)

$P_t =$ Rotate(P_s, θ)
$M_t =$ MinkowskiSum(P_t, Q)
return (M_t)

5.3 Dynamic Minkowski sums (DYMSUM)

In this section, we describe the details of DYMSUM. Our goal is to take advantage of the correspondences between M_s and M_t that are completely ignored by Algorithm 5.2.1. Let us

consider the Gaussian map again. M_s is computed by overlaying $g(P_s)$ and $g(Q)$. To obtain M_t, we need to find out which vertices in $g(P_s)$ are moved to another face in $g(Q)$ and determine whether the edges of $g(P_s)$ intersect or stop intersecting with the edges of $g(Q)$ after rotating P_s to P_t . This is exactly what DYMSUM does. That is, DYMSUM first determines these changes in the overlay introduced by the rotation, and then corrects the errors to generate the new Minkowski sum M_t. Therefore, the Minkowski sum M_t is composed of two types of facets: (1) the facets from M_s that still satisfy the aforementioned criteria after rotation and (2) the facets that are created due to the errors.

A sketch of DYMSUM is shown in Algorithm 5.3.1. In contrast to the brute-force method, DYMSUM is sensitive to the amount of rotation. That is, when θ is smaller, there will be fewer errors in the Gaussian map overlay. In this case, DYMSUM will likely take less time to compute the result than the naïve method. In the rest of this section, we will discuss how the errors are determined (Section 5.3.1) and how to correct these errors (Sections 5.3.2 and 5.3.3).

Algorithm 5.3.1: DYMSUM(M_s, P_s, Q, θ)

$P_t =$ Rotate(P_s, θ)
$E_t =$ FindErrors(M_s, P_t, Q)
$M_t =$ CorrectErrors(E_t, M_s, P_t, Q)
return (M_t)

5.3.1 Find Errors

There are two types of errors, fv-errors and ee-errors, corresponding to fv-facets and ee-facets, respectively. If a pair of features was compatible and becomes incompatible after the rotation of P, we call this pair an error.

Before we talk about how these errors can be identified, we will first show the relationship between the fv-errors and the ee-errors. Theoretically, the complexity of the Minkowski sum

is $O(mn)$ and there can only be $O(m+n)$ fv-facets. Therefore, the number of fv-facets can be far smaller than the number of ee-facets. Moreover, it is easy to show that no ee-errors can occur if there are no fv errors.

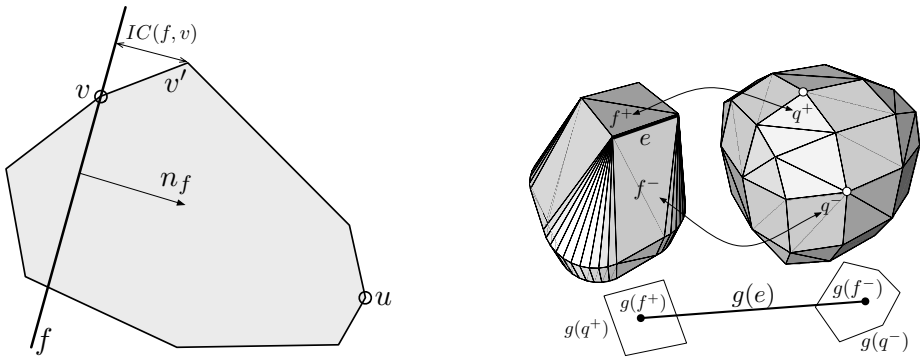
Theorem 1. **fv-errors and ee-errors must coexist.**

Proof. We first show that if there is an ee-error, there must be an fv-error. Let e and e' be a pair of edges that are compatible before rotation and become an ee-error after rotation. When e and e' are compatible, $g(e)$ and $g(e')$ must intersect and, after rotating P, $g(e)$ and $g(e')$ no longer intersect. This means at a certain point during the rotation, an end point of $g(e)$ must cross $g(e')$ or vice versa. When a point v crosses the edge $g(e)$, v changes the face with which it is associated from one side of $g(e)$ to the other side of $g(e)$. This change indicates that there must be an fv-error.

We then show that if there is an fv-error, there must be an ee-error. If a facet f of P and a vertex v of Q become an fv-error, we know that f now must be compatible with some other vertex $v' \neq v$ of Q. As a result, an edge $g(e)$ incident to $g(f)$ must be moved (or deformed) with $g(f)$. Since the faces in $g(Q)$ are convex and $g(e)$ cannot intersect with a segment more than twice, $g(e)$ must intersect with some new edges of Q when $g(f)$ moves from $g(v)$ to $g(v')$. This indicates that there must be at least one ee-error.

Therefore, fv-errors and ee-errors must coexist. □

Based on Theorem 1, we can find all errors by first exhaustively checking all fv-facets in M_s (the Minkowski sum before rotation) to find fv-errors. Then we use these fv-errors to identify all ee-errors. That is, if there are no fv-errors found, then we can immediately conclude that there are no ee-errors as well. Otherwise, the ee-errors must occur at the edges incident to the vertices involved in the fv-errors. Thus, it is clear that finding all fv-errors will take $O(m+n)$ time.



(a) degree of incompatibility (b) correct an ee-error

Figure 5.2: (a) A 2-d drawing shows the definition of $IC(f, v)$ and its witness $\overline{vv'}$. Using gradient descent, we will find the compatible vertex u for f. (b) Determine the associated edges for edge e.

5.3.2 Correct fv-errors

For each fv-error, we perform a gradient descent to compute a new fv-facet. More specifically, given a facet/vertex pair, we can measure the *degree of incompatibility* of the pair and attempt to iteratively minimize the incompatibility until a compatible pair is found.

Let f be a facet of P and v be a vertex of Q. When f and v are compatible, all the edges that are incident to v must be *below* or on the half-plane supported by f. When f and v are incompatible, we can define the *degree of incompatibility*

$$IC(f, v) = \max\{\mathrm{d}(e, f) \mid e \in E_v\} \ ,$$

where E_v is a set of edges incident to v and $\mathrm{d}(e, f)$ is the longest Euclidean distance from any point on e to f. We say an edge e is the witness of the incompatibility if $\mathrm{d}(e, f)$ is $IC(f, v)$. Fig. 5.2(a) illustrates an example of fv-error and $IC(f, v)$.

In order to find the compatible pair, we find the witness of the incompatibility e, and replace v with the other end point $v' \neq v$ of e, and repeat this until f and v become compatible. In Fig. 5.2(a), this vertex is u. Since Q is convex, this procedure must be able

to find a vertex of Q such that all of its incident edges are below f, therefore, will always terminate.

This process is equivalent to finding an extreme point at the outward normal direction of f and therefore can be done in $O(\log n)$ time if Q has n vertices.

5.3.3 Correct ee-errors

After all the fv-errors are corrected, the incident edges associated with these fv-errors are marked as ee-errors. Let e be such an edge from P and let f^- and f^+ be the facets in P incident to e. Our goal is to find the edges of Q that are compatible with e. An exhaustive search for compatible edges will certainly be slow. Fortunately, we can find the compatible edges using the results from fv-facets. That is, since we know that the incident facets f^- and f^+ both have the compatible vertices q^- and q^+ of Q, we can find the compatible edges for e using q^- and q^+. The relationships between e, $f^\pm$ and $q^\pm$ are shown in Fig. 5.2(b).

More specifically, if we overlay the Gaussian map $g(e)$ of e with $g(Q)$, $g(e)$ will intersect a set of faces in $g(Q)$ and the end points of $g(e)$ are inside $g(q^-)$ and $g(q^+)$. See the bottom of Fig. 5.2(b). If we can determine the rest of the faces intersected by $g(e)$, we can find the compatible edges for e. We further know that these faces form a connected component between $g(q^-)$ and $g(q^+)$, thus the compatible edges for e must be on the boundary of these faces. To find these Gaussian faces, we start from $g(q^-)$, and find an incident edge e' of $g(q^-)$ that is compatible with e. It is obvious that e' must exist unless $q^- = q^+$. From e', we replace q^- with the vertex $x' \neq q^-$ incident to e', and repeat the process until $q^- = q^+$.

The computation time is equal to the sum of the degree of vertices of Q visited during the search process.

5.4 Experimental results

In this section, we show that the computation time of DYMSUM is more efficient than the traditional approach (Algorithm 5.2.1) and is indeed sensitive to the amount of rotation

applied to P. In our experiments, the polyhedron P rotates using a sequence of random quaternions. Each quaternion is applied to P for a random period of time. All the computation times that we show below are obtained by averaging over 100 random rotations. All the experiments are performed on a machine with Intel CPUs at 2.13 GHz with 4 GB RAM. Our implementations are coded in C++.

Figures 5.3 and 5.4 show 13 models that we use in the first two experiments. Many of these models are from [26] and can be obtained from the authors' website. Theoretically, DYMSUM works with polyhedra tessellated with arbitrary polygons, but in our current implementation DYMSUM only takes triangulated polyhedra. Therefore, all the models in Figures 5.3 and 5.4 are triangulated.

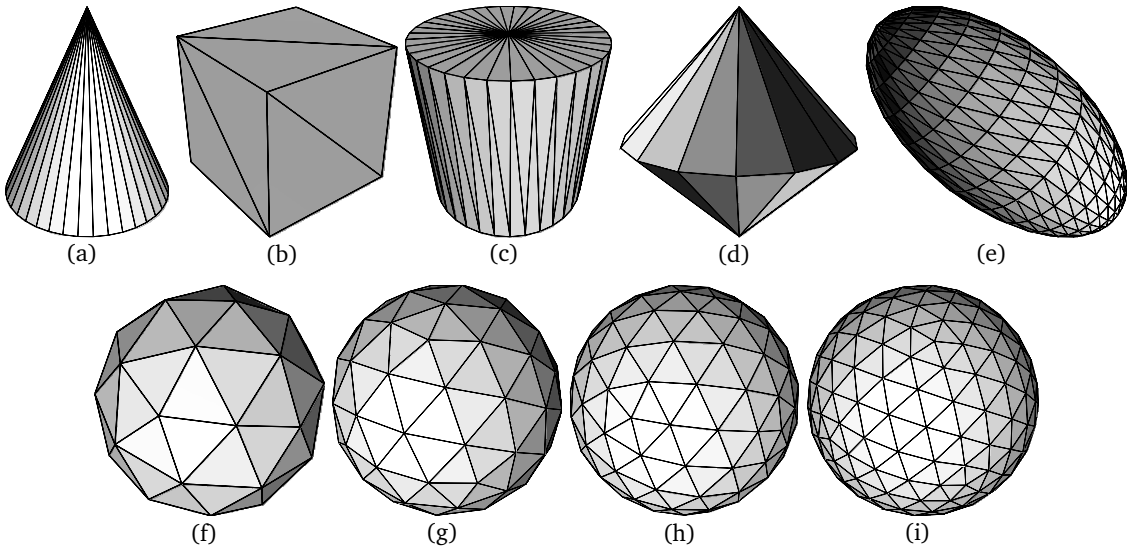


Figure 5.3: Models used in the experiments. (a) cone, 78 facets (b) cube, 12 facets (c) cylinder, 140 facets (d) dioctagonal dipyramid (DD), 32 facets (e) ellipse, 960 facets (f) geodesic sphere 1 (GS1), 80 facets (g) GS2, 180 facets (h) GS3, 320 facets (i) GS4, 500 facets

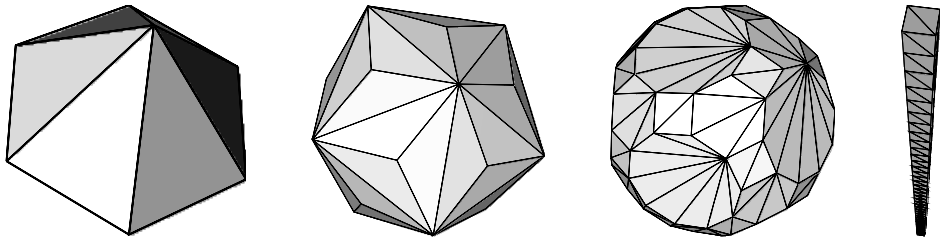


Figure 5.4: Models used in the experiments. (a) hexagonal pyramid (HP), 10 facets (b) triakis icosahedron (T), 60 facets (c) truncated icosidodecahedron (TI), 236 facets (d) v-rod, 324 facets.

5.4.1 Experiment 1: Dymsum vs. Brute-force method

In Table 5.1, we compare the proposed method, DYMSUM, to a brute-force method (Algorithm 5.2.1) that re-computes the Minkowski sum in every time step. The brute-force method checks the compatibility of all facet-vertex and edge-edge pairs every time that P rotates. The values in the table are t_d/t_{bf}, where t_d and t_{bf} are the (averaged) updating or re-computing times for DYMSUM and the brute-force method.

From Table 5.1, it is clear that DYMSUM is always faster than the brute-force method. Even for very simple cases, such as cone$\oplus$HP, DYMSUM is at least 8 times faster. For more complex examples, such as ellipse$\oplus$ellipse, DYMSUM is about 176 times faster than the brute-force method.

5.4.2 Experiment 2: Computation time vs. Rotational resolution

In this experiment, we study the computation time of DYMSUM with respect to the rotational resolution of P. Our goal is to show that, in contrast to the brute force approach, DYMSUM is in fact sensitive to the magnitude of the rotation. In the problem of motion planning, this resolution defines the number of slices in mapping the configuration space. In the physically-based simulation, this value defines the number of collision detections and penetration depth estimations per second. Fig. 5.5 shows the results obtained using DYMSUM. Notice that the x

Table 5.1: The speedup of dymsum using the models in Figs. 5.3 and 5.4. The values in the table are t_d/t_{bf}, where t_d and t_{bf} are the computation times for DYMSUMand the brute-force method.

cone	9.26						
cube	8.58	10.50					
cylinder	12.83	10.31	18.73				
DD	9.82	9.85	13.33	10.93			
ellipse	24.97	14.50	47.51	21.67	**176.34**		
GS1	15.09	11.21	23.66	15.39	51.38	25.23	
GS2	19.73	12.15	32.96	17.95	86.99	33.16	52.73
GS3	22.92	12.42	40.45	19.63	120.23	38.88	65.53
GS4	24.50	12.70	45.22	20.19	146.15	44.21	73.89
HP	**8.02**	9.32	9.59	9.25	13.19	9.75	10.84
T	14.50	11.16	20.66	14.05	35.95	20.33	26.01
TI	21.30	13.87	35.55	19.53	91.48	38.02	56.86
v-rod	20.30	18.31	34.00	21.92	88.30	46.53	65.46
	cone	cube	cylinder	DD	ellipse	GS1	GS2

Table 5.2: Speedup of dymsum cont.

GS3	86.26					
GS4	100.15	121.26				
HP	11.58	11.51	9.00			
T	29.21	31.83	10.28	16.17		
TI	69.25	78.26	12.78	28.08	63.45	
v-rod	77.73	83.05	15.76	34.23	67.24	123.75
	GS3	GS4	HP	T	TI	v-rod

axis is in logarithmic scale.

The x axis of Fig. 5.5 is the number of steps for P to make a full rotation. For example, when $x = 500$, P will take 500 steps to rotate 360° degree. That is, P rotates $\pi/250$ around a random axis every step. Therefore, when x is large, the changes in the Minkowski sum will be small. From the figure, we can see that the computation time drops quickly around $x = 500$ and then stabilizes below the 0.5 millisecond mark. In Experiments 1 and 3, we set $x = 500$.

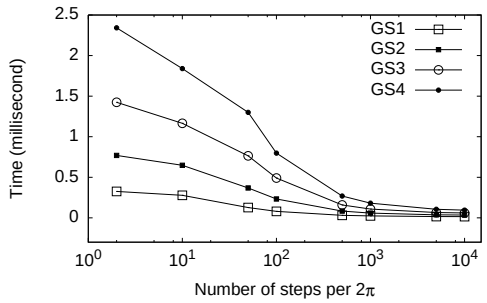
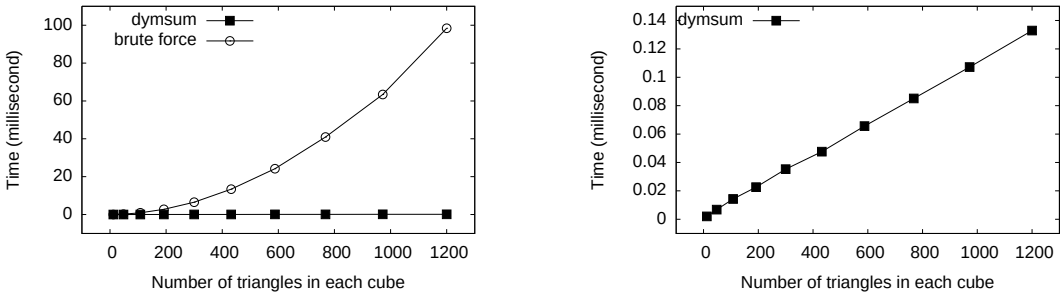


Figure 5.5: Computation time at different rotational speeds. More steps per 2π means slower rotational speed.



(a) DYMSUM vs. brute force (b) DYMSUM only

Figure 5.6: Computation times of DYMSUM and brute force of two identical cubes. The numbers of triangles in the cubes are 12, 48, 108, 192, 300, 432, 588, 768, 972 and 1200.

5.4.3 Experiment 3: Computation time vs. Input size

In this last experiment, we study the relationship between the computational time and the input size. We use a $10 \times 10 \times 10$ cube with different numbers of triangles tessellated on the surface. Fig. 5.6 shows that the computation time of the brute-force method increases rapidly while that of DYMSUM stays almost constant. When we show DYMSUM's computation time along in Fig. 5.6(b), DYMSUM's computation time is increased linearly along with the size of the cubes.

Recall that the complexity of a Minkowksi sum of two convex shapes is $O(mn)$, however the number of the fv-facets is $O(m+n)$. Therefore a large portion of the Minkowksi sum is composed of the ee-facets. In our experiment, we see a linear increase in computation time.

We speculate that only a few errors occur at each step and most of the computation time is spent on verifying and updating the compatibility of the fv-facets.

Chapter 6: Dynamic scaling of arbitrary polygons and convex polyhedra

6.1 Introduction

The development of a method for the operation of rotation, which is global on the input it acts on, leads us naturally to explore the idea of other global operations. In particular, we consider the case of scaling an object, either uniformly or non-uniformly along its axes. Scaling of an object P is simply the transformation of each vertex $v = (v_1, v_2, ..., v_d)$ in P to a new position $v' = (s_1v_1, s_2v_2, ..., s_dv_d)$, where $v \in \mathbb{R}^d$ and $s_i \in (0, \infty)$. Uniform scaling consists of the special case where $s_1 = s_2 = ... = s_d$.

In [13] we propose two methods for dynamic Minkowski sums of inputs under scaling. We base our first method on the Minkowski sum methods in our previous work [12, 40], which computes the reduced convolution of the two polygons or two polyhedra. In 2D, the reduced convolution is a subset of the full convolution which omits contributions from the reflex vertices of the input polygons. Reflex vertices are vertices whose interior angles are greater than 180 degrees. Two reduced convolutions are illustrated in Fig. 6.1.

Given this reduced convolution, denoted by $P \widehat{\otimes} Q$, we show that there is an efficient way to update 2D Minkowski sums under scaling regardless of the convexity of the input models (Section 6.2). The set of intersecting faces in the Minkowski sum changes only at a finite set of scale values. Scaling of an object P by a scale factor s is simply the transformation of each vertex $v = (v_1, v_2, ..., v_d)$ in P to a new position $v' = (s_1v_1, s_2v_2, ..., s_dv_d)$, where for $v \in \mathbb{R}^d$ and $s_i \in (0, \infty)$. Uniform scaling consists of the special case where $s_1 = s_2 = ... = s_d$.

Our goal is to develop an algorithm that can efficiently *update* the Minkowski sum when the objects are scaled without recomputing the Minkowski sum from scratch. The main

challenge comes from the fact that the Minkowski sum can change drastically when the underlying objects are scaled. For example, in Fig. 6.1, when the disc is scaled to twice its original size, not only the geometry but also the topology of the Minkowski sum changes.

We present two *exact, output-sensitive algorithms* whose computation time depends on the amount of scale and therefore depends on the number of changes to the Minkowski sum due to scaling.

We base our first method on the Minkowski sum methods in our previous work [12, 40], which computes a *reduced convolution* of the two polygons or two polyhedra. In 2D, the reduced convolution is a subset of the full convolution which omits contributions from the reflex vertices of the input polygons. Reflex vertices are vertices whose interior angles are greater than 180 degrees. Two reduced convolutions are illustrated in Fig. 6.1.

Given this reduced convolution, denoted by $P \widehat{\otimes} Q$, we show that there is an efficient way to update both 2D Minkowski sums under scaling regardless of the convexity of the input models (Section 6.2). The set of intersecting faces in the Minkowski sum changes only at a finite set of scale values.

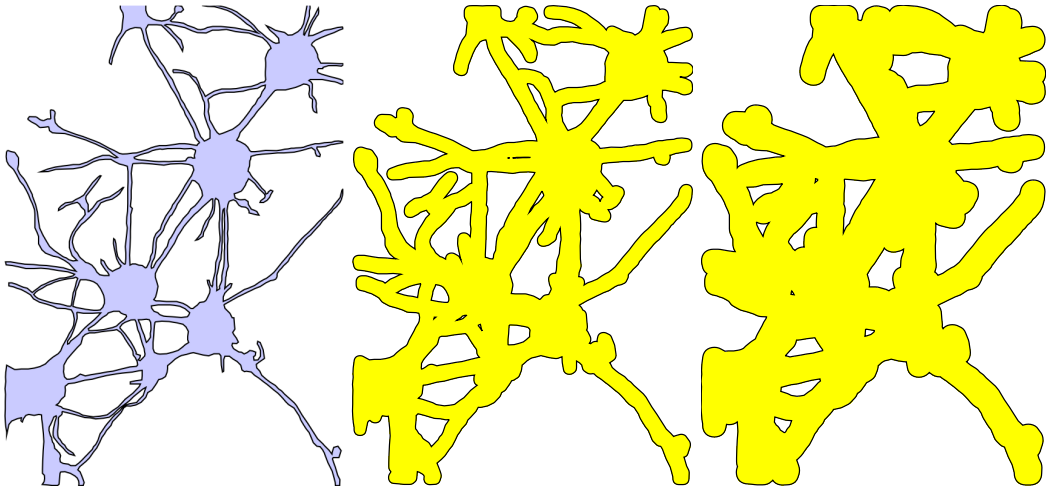
We show that this method can be extended naturally to 3D (in Section 6.3). However, we also show that it is not practical to do so due to time and space complexity constraints as the convolution in 3D is much more complex. To address this, in Section 6.4, we introduce the second method, which dynamically identifies compatibility errors in the convolution for convex polyhedra and corrects these errors without precomputing when they will occur. We show that this method supports non-uniform scaling, provides significant speed improvements over brute force, and does not encounter the complexity constraints of precomputation.

In addition to reusing computations for the obvious purpose of increasing the computation efficiency, in Section 6.6 we also demonstrate that our method can be extended to answer the query, "Given the specification of manufacturing tolerance, what are the largest and smallest scales of P for which a given path is valid?" This query is useful in rapid prototyping, for determining the necessary scale for parts to fit into an assembly. With this information we

can check whether there exists a scale for a part P that can guarantee it fits into the assembly properly. The proposed method also finds unusual applications in shape decomposition, where it can be used to identify structural features based on multi-scale convolutions.

This work [13] is the first to consider dynamic Minkowski sums under scale and our results are encouraging. However, there are two main limitations for the first method. First, it is only able to handle uniform scaling, as it relies on the idea that the facet normals do not change under scaling. Secondly, it is impractical in more than two dimensions due to its large time and space complexities. In three or more dimensions, we quickly run up against the curse of dimensionality. The main limitation of the second method is its inability to handle non-convex inputs, since it relies on the principle that each facet in the convolution will be associated with at most one vertex in the input models.

6.2 Uniform Scaling in 2D



(a) *input* (b) $s = 1$ (c) $s = 2$

Figure 6.1: Comparison of the Minkowski sums [(b) and (c)]. The neuron polygon shown in (a) has 18 holes and 1,815 vertices and the disc is represented as a polygon with 32 vertices. The topology of the Minkowski sum changes between scales.

We seek to update the Minkowski sum of polygons P and Q transforming under uniform scales. We assume without loss of generality that P scales uniformly and Q remains fixed. Since the Minkowski sum is commutative, scaling Q can be done identically by simply swapping P and Q in the input order. The uniform scaling operation $P' = sP$ generates P' such that for every point $v \in P$, $sv \in P'$. A key observation is that when P scales by s, the outward normals of all its line segments remain the same, and so scaling of P does not change the combinatorial structure of the reduced convolution (RC); only the length and position of the segments in RC change.

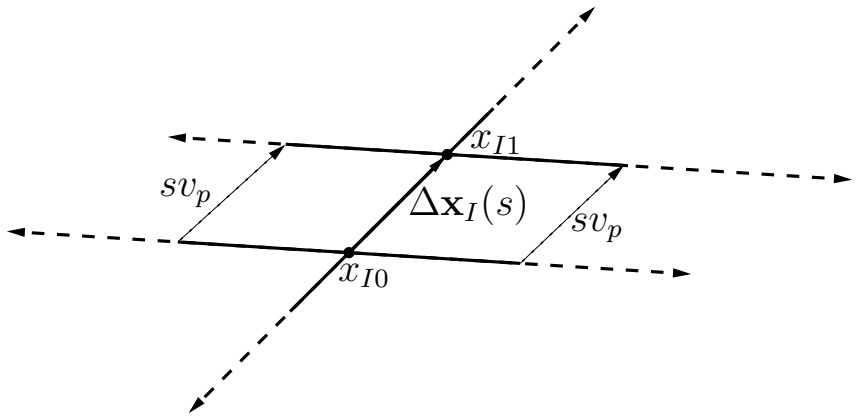
Even though the combinatorial structure of the RC does not change when P is scaled, the outer boundary of the Minkowski sum will change. In particular, the set of edges (2D) or facets (3D) which intersect with each other may change–some intersections may be deleted while others may be added. We call these additions and deletions *critical events*.

To use these critical events, we compute the ranges of s within which intersections between edges actually occur. We can then add or delete intersections by stepping through the critical events between two values of s. When we finish adding and deleting intersections in this way, we can correctly and efficiently update by considering only those intersections between line segments that are valid for the given s.

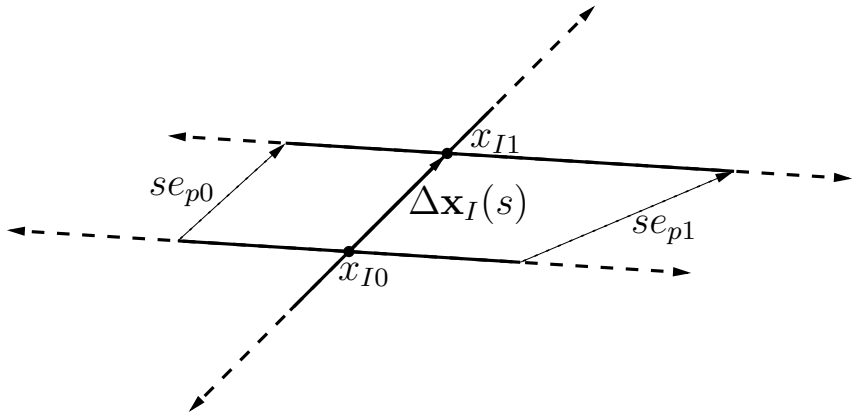
6.2.1 Tracking the intersection values in 2D

Consider a single edge of the convolution, a. Suppose a is formed from the contribution of an edge of P, e_p, and a vertex of Q, v_q. Then $a = e_p + v_q$, that is, e_p translated by v_q. When P scales, only e_p changes, so $a(s) = se_p + v_q$. Because such edges are scaled as P is scaled, we call them **S-edges**. Note that S-edges will also be translated in the updated convolution. Similarly, if a is formed from an edge of Q, e_q and a vertex of P, v_p, then $a = e_q + v_p$, and $a(s) = e_q + sv_p$, since only P scales. We call such edges **T-edge** since as P is scaled, they are translated but not scaled. Fig. 6.2 illustrates both types of edges.

Now consider two line segments a and b in the convolution, and let their endpoints be



(a) T-edge



(b) S-edge

Figure 6.2: In (a) an edge from Q and a vertex from P form a T-edge in the convolution, so the segment only translates as s changes. In (b) an edge from P and a vertex from Q form an S-edge edge, so the segment also scales as it translates. In both cases the underlying lines translate.

(a_0, a_1) and (b_0, b_1), respectively. Let the lines containing a and b be l_a and l_b, respectively. We parameterize the intersection point of l_a and l_b over $s \in (0, \infty)$. To do this, we consider the parameterized forms of l_a and l_b:

$$l_a(r) = a_0 + r(a_1 - a_0) = a_0 + r\mathbf{v}$$

$$l_b(t) = b_0 + t(b_1 - b_0) = b_0 + t\mathbf{w}\ ,$$

where $\mathbf{v} = a_1 - a_0$ and $\mathbf{w} = b_1 - b_0$.

Observation 4. *Let $\boldsymbol{u} = a_0 - b_0$. Then the intersection point of l_a and l_b is at*

$$r_I = \frac{\boldsymbol{v}_y \boldsymbol{u}_x - \boldsymbol{v}_x \boldsymbol{u}_x}{\boldsymbol{v}_x \boldsymbol{w}_y - \boldsymbol{v}_y \boldsymbol{w}_x} ,$$

and the coordinate of that intersection is $x_a(r_I) = a_0 + r_I \boldsymbol{v}$.

The parameterized form lets us find the critical events we are looking for.

6.2.2 Finding critical regions

Let $x(s)$ be the intersection point between the line l_a containing a and the line l_b containing b when P is scaled by a factor of s. $x(s)$ is a parameterized line as well. Let the intersection point computed above be x_0. This intersection is computed at the base scale value, $s = 1$. Therefore, $x(1) = x_0$. Then we can define $x(s) = x_0 + (s-1)\sigma$ for a slope vector σ. It is easy to recompute the line segments at some other scale, s', and compute the intersection of the new lines to find $x(s') = x_0 + (s'-1)\sigma$, and so $\sigma = \frac{x(s') - x_0}{s'-1}$. We choose $s' = 2$, since in this case, $s' - 1 = 1$ and so $\sigma = x(2) - x(1)$, which eliminates the division.

Assume that a and b do not intersect. As P scales, if a and b start to intersect, they must do so first at an endpoint of either a or b. Similarly, when a and b separate, their final point of contact will be at an endpoint of either a or b. The values of s where these initial and final points of contact for some set of edges occur are the critical events.

We begin by parameterizing the endpoints of a over s, as $a_0(s), a_1(s)$. We wish to find the values of s for which $x(s) = a_0(s)$ and $x(s) = a_1(s)$ that is, the scale factors for which the intersection of the two edges is precisely the endpoint of one of them.

Critical region for S-edge

Suppose that a is an S-edge in the convolution is formed from an edge, e_p in P and a vertex v_q in Q. Then:

$$a_0(s) = e_{p0}s + v_q$$

$$a_1(s) = e_{p1}s + v_q \ .$$

Setting $x(s) = a_0(s)$ we obtain $x_0 + (s-1)\sigma = se_{p0} + v_q$, $x_0 - \sigma - v_q = s(e_{p0} - \sigma)$. This is an equation of the form:
$s\alpha = \beta$, where $\alpha = \langle \alpha_1, \alpha_2 \rangle$ and $\beta = \langle \beta_1, \beta_2 \rangle$. Since s is scalar in uniform scaling, this is an overdetermined system of equations; there is no solution for s if the system is inconsistent. Otherwise, we obtain a possible bound on s values for contact with the $a_0(s)$. We can proceed to totally bound the s interval where the intersection lays on the segment a by computing the same bound for $a_1(s)$, which is of the same form.

Critical region for T-edge

But what about convolution edges formed from an edge e_q of Q and a vertex v_p of P? Assume, without loss of generality, that b is such an edge, and so $b = e_q + v_p$. Then its endpoints are defined by $b_0(s) = e_{q0} + sv_p, b_1 s = e_{q1} + sv_p$.

Again we seek to bound the intersection in this case by finding s such that $x(s) = b_0(s)$ and $x(s) = b_1(s)$. As above, $x(s) = x_0 + (s-1)\sigma$, and so the form of these equalities are: $x_0 + (s-1)\sigma = e_{qi} + sv_p$. This is an overdefined system of the same form as above, subject to the same results.

Critical region of a segment pair

To obtain the intersection interval for the segment pair, we just take the intersection of the intervals computed above, $[\max\{B_{a_0}, B_{b_0}\}, \min\{B_{a_1}, B_{b_1}\}]$ where B_u is the s boundary for the endpoint u.

We can compute these intervals for all pairs of line segments as a pre-processing step in $O(n^2)$. Using this, we can compute a list of critical events–s values when intersections are introduced into the arrangement or removed from it. This allows us to update only those intersections at each scaling operation which are relevant. Because computing the intersections of the arrangement is the largest bottleneck in the reduced convolution method, this promises significant speed advantages over recomputing the arrangement from scratch every time s changes.

Update arrangement for a given scale factor s

In order to update the arrangement, we produce an array of critical events: pairings of s values with a list of insertions or deletions since the last event produced by the intervals for pairs of line segments. Given an initial scale value s_0 and its arrangement, and a final scale value s_{final}, we can step through the structure, deleting and adding nodes in the arrangement as necessary at each event until we reach an event such that $s_{event} > s_{final}$, at which point we update all of the remaining nodes according to the above equations.

6.3 Uniform scaling in 3D

In 2D, events are defined by the contact of two edges. Similarly, in 3D, events are defined by the contact of two *faces*. There are two possible cases for these events in 3D: a vertex of one face comes into contact with any part of another face, or an edge of one face comes into contact with an edge of the other face without any vertex contacts. We call these *vertex events* and *edge events*, respectively.

6.3.1 Finding vertex events

Computing vertex events is a relatively straightforward extension of the 2D case. Given two faces, f_1 and f_2, we consider each edge e of f_1 independently, and we parameterize the line l containing e and the plane p containing f_2, and compute the intersections at $s = 1$ and $s = 2$ as in the 2D case. Uniform scaling still only causes linear motion in 3D, and so computing where the vertex contact occurs is fundamentally identical to the 2D case.

We can again compute $\sigma = x(2) - x(1)$, and parameterize the intersection point across s as $x(s) = x_0 + (s-1)\sigma$. We then set the intersection point equal to the parameterized endpoints $e_1(s)$ and $e_2(s)$ to find candidate events, identically to the 2D case. The equation $x(s) = e_i(s)$ is overdetermined. There are three equations and only one free variable, similar to the 2D case, and so there may be no consistent solution again – this will occur when the edges are parallel or skew.

If there is a consistent solution we check and make sure these intersection points also lay in the face f_2 just as one would ensure that the contacts in 2D lay on the line segment and not just the line. The vertex events between the edges of f_2 and f_1 are found in the same way.

6.3.2 Finding edge events

Computing edge events is slightly more complicated. We consider two edges, e_1 and e_2, and the lines which contain them, l_1 and l_2 respectively. We cannot simply check for an intersection point since it is most likely that l_1 and l_2 are skew. Instead, faces f_1 incident to e_1 and f_2 incident to e_2, and their supporting planes, p_1 and p_2 respectively. We are interested in the intersection x_1 between l_1 and p_2 and the intersection x_2 between l_2 and p_1. Then l_1 and l_2 will intersect precisely when $x_1 = x_2$.

A line l which contains an edge e with endpoints e_a and e_b can be parameterized as $e_a + (e_b - e_a)t$, and the plane p which contains triangle t with vertices t_a, t_b, t_c can be parameterized as $t_a + (t_b - t_a)u + (t_c - t_a)v$. We set these equal to each other and simplify,

yielding

$$e_a - t_a = \begin{bmatrix} e_a - e_b \\ t_b - t_a \\ t_c - t_a \end{bmatrix}^T \begin{bmatrix} t \\ u \\ v \end{bmatrix} .$$

We have five vertices from the convolution in this equation, e_a, e_b, t_a, t_b, t_c, each of which is parameterized in the scale domain by its P and Q components, $v = v_q + sv_p$. Substituting, we obtain expressions for t, u and v parameterized by s:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \left(\begin{bmatrix} e_{aq} - e_{bq} + s(e_{ap} - e_{bp}) \\ t_{bq} - t_{aq} + s(t_{bp} - t_{ap}) \\ t_{cq} - t_{aq} + s(t_{cp} - t_{ap}) \end{bmatrix}^T \right)^{-1} (e_{aq} - t_{aq} + s(e_{ap} - t_{ap})) .$$

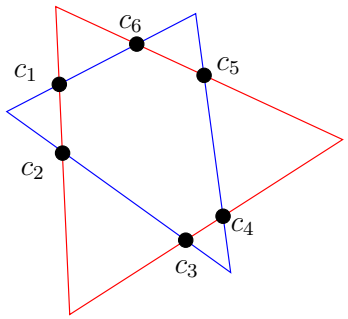


Figure 6.3: Degenerate ee contacts for parallel faces, contacts are marked as c_1 through c_6. Notice that no vertices are involved in this event.

When we solve this for l_1 and p_2 as well as l_2 and p_1, we obtain two expressions in s for the intersections of the line and the plane, which we can set equal to each other and solve for s.

Parallel faces When f_1 and f_2 are parallel but not always coplanar, then

$$\begin{bmatrix} e_a - e_b \\ t_b - t_a \\ t_c - t_a \end{bmatrix}^T ,$$

will be singular, and so we will not be able to detect edge events in the usual way. If any vertex of either face is involved in the contact, then we will correctly detect these events as vertex events, but it is possible that this isn't the case. Figure 6.3 shows a situation in which two coplanar faces have no vertex contacts. In order to determine whether an event will occur, we must find the scale value when the faces are coplanar, and then check the edges of the faces for intersections at that scale. Fortunately, the detection for vertex events is robust enough that it does not need to deal with parallel faces as a special case.

To determine the value of s where the faces are coplanar, we pick an arbitrary vertex v on f_1 and proceed as though we were detecting a vertex event with f_2. However, we do not need to check if $v(s_{event})$ is inside of f_2, since we are only interested in the value of s where v hits p_2–this will be the value of s at which the faces are coplanar.

Because of this, we do not need to perform the usual edge event detection at all when dealing with parallel faces. For these faces, we find the scale of contact from the vertex event detection step. If there are no vertex events, we check for edge intersections to see if there are any edge events.

Coplanar faces Finally, in some cases, f_1 and f_2 will always be coplanar. In these coplanar cases, the contact possibilities for f_1 and f_2 reduce to the 2D case, in which edge events are impossible. Therefore, we can use the logic from the 2D scaling directly in order to detect the events for the coplanar faces by transforming f_1 and f_2 into the xy-plane, and applying $2D$ vertex event detection.

6.4 Non-uniform scaling for convex polyhedra

In three dimensions, the worst-case complexity of the event space is $O(m^4n^4)$. The memory necessary to store the event structure and the time necessary to compute it rapidly become overwhelming. While there are several practical strategies for mitigating memory issues, including narrowing the scaling domain to a smaller window and disk swapping, the event space must be computed for all pairs of faces, resulting in worst-case time complexity. Enumerating all critical events also makes the extension to non-uniform scaling almost impossible. As a result, we look to a method with less pre-computation.

Similar to our strategy for handling rotation [10], we propose to dynamically repair the Minkowski sum after scaling. To do so, we define $errors$ in the transformed convolution to be convolution facets constructed from either *vertex-face* (vf) pairs or *edge-edge* (ee) pairs that are no longer compatible, called fv-errors and ee-errors, respectively. Our method works simply by correcting the errors introduced by the scaling. Correcting the fv-errors involves gradient descent on the *degree of incompatibility* until the vertex and face pair become compatible. Let f be a facet of P and v be a vertex of Q. When f and v are compatible, all the edges that are incident to v must be *below* or on the half-plane supported by f. When f and v are incompatible, we can define the *degree of incompatibility*: $IC(f, v) = \max\{\mathrm{d}(e, f) \mid e \in E_v\}$, where E_v is a set of edges incident to v and $\mathrm{d}(e, f)$ is the longest Euclidean distance from any point on e to f. We say an edge e is the witness of the incompatibility if $\mathrm{d}(e, f)$ is $IC(f, v)$. Fig. 5.2(a) illustrates an example of fv-error and $IC(f, v)$.

We have already shown that any ee-errors must coexist with fv-errors (see Theorem 1). Since, for convex pairs, there are only $O(m + n)$ vf-facets, but $O(mn)$ ee-facets, we can increase the repairing speed by first checking all fv-facets for errors. Using the fv-errors as starting points, we then form chains of ee-errors connected to the vf-error.

More specifically, let e be such an edge from P involved in an ee-error, and let f^- and f^+ be the facets in P incident to e. Assume that the facets f^- and f^+ both have the compatible vertices q^- and q^+ of Q. If we overlay the Gaussian map $g(e)$ of e with $g(Q)$,

$g(e)$ will intersect a set of faces in $g(Q)$ and the end points of $g(e)$ are inside $g(q^-)$ and $g(q^+)$. See the bottom of Fig. 5.2(b). If we can determine the rest of the faces intersected by $g(e)$, we can find the compatible edges for e. We further know that these faces form a connected component between $g(q^-)$ and $g(q^+)$, thus the compatible edges for e must be on the boundary of these faces. To find these Gaussian faces, we start from $g(q^-)$, and find an incident edge e' of $g(q^-)$ that is compatible with e.

Because both P and Q are convex, the faces f_1 and f_2 incident to e will each be compatible with precisely one vertex, v_1 and v_2 respectively. The edges with which e is compatible must form a path from v_1 to v_2 along the surface of the model. We check each of v_1's incident edges to find the edge with which e is compatible. We then replace v_1 with the vertex at the other end-point of the compatible edge, and repeat, ignoring the incident edge which has already been found to be compatible. When we find v_2 by this path, we have enumerated all of the compatible vertices. The complexity of this update is $O(|E|)$, where E is the set of edges incident to at least one vertex in the path.

In convex pairs, each face of P will be paired with precisely one vertex of Q and vice-versa. We depend on this relationship in order to correctly identify errors. An extension to non-convex inputs depends on the existence of a *convex map*–a convex polyhedron whose Gaussian map is identical to the arrangement of the Gaussian map of the non-convex input. Discussion of the convex map is outside the scope of this paper.

6.5 Results

In this section, we reproduce our experimental results from [13] to demonstrate the performance of the proposed methods. Our implementations are based on the publicly available implementation from our previous work in 2D [12] and 3D [10]. All experiments reported in this section are performed on a machine with Intel CPUs at 2.13 GHz with 4 GB RAM and our implementation is coded in *C++*, using *GNU MPFR* and the *MPFRC++* wrapper for high precision arithmetic.

6.5.1 Results from uniform Scaling in 2D

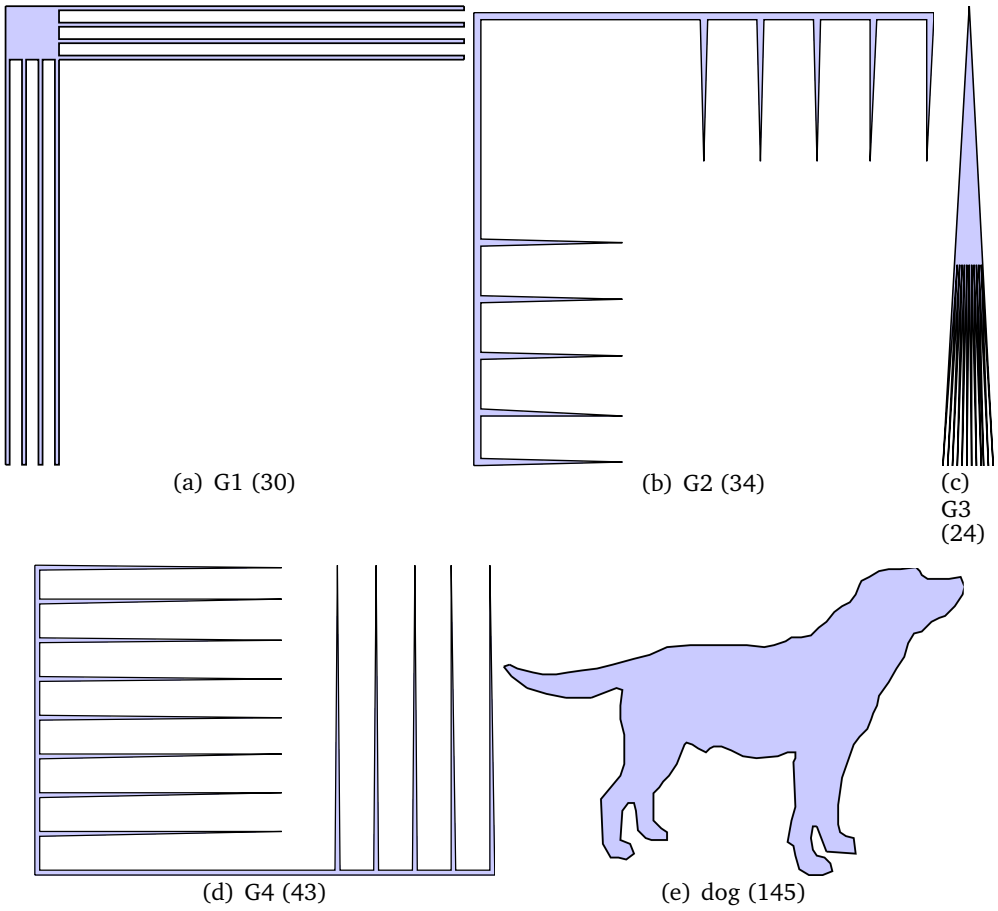
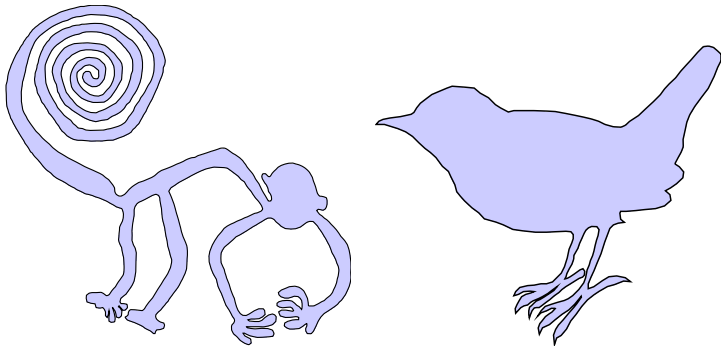


Figure 6.4: Models used in the experiments. The figure shows the names and the sizes of the polygons. Note that (c) is G3 (24).

Table 6.1 shows average speed-up factors for several model pairs in Fig. 6.4 across the scaling domain $[0.1, 10]$ in increments of 0.1 scale factor. Speed-up is computed as $\frac{t_{static}}{t_{dynamic}}$ over 100 random re-scalings. Our method is always faster than recomputing from scratch and is on average 150.682 times faster. The largest speedup is over two orders of magnitude (from $g1 \oplus g4$) and even on model pairs with high event density ($dog \oplus g2$ in Table 6.1), the

Table 6.1: Average speed-up of the 2D enumerative dynamic update algorithm over 100 random re-scalings. Models in the first column are the model being scaled in each experiment, while models in the first row are the static (unscaled) model. The bolded figures are the lowest and highest speed-up values achieved.

	dog	$g1$	$g2$	$g3$	$g4$
dog	28.911	115.651	**5.592**	32.408	6.538
$g1$	115.651	131.150	432.069	533.141	**473.282**
$g2$	15.223	313.710	298.038	161.712	88.916
$g3$	112.354	152.790	171.825	34.347	79.362
$g4$	14.519	201.861	45.220	80.230	122.565



(a) Input P, monkey (1204) (b) Input Q, bird (275)

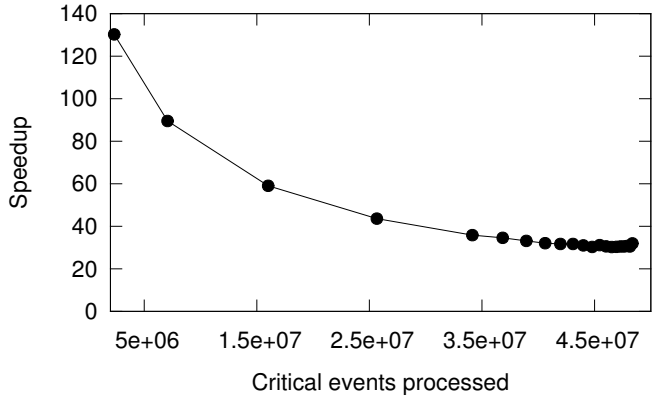
Figure 6.5: Input models for the graphs shown in Figure 6.6

proposed method is faster though the improvement here is more marginal.

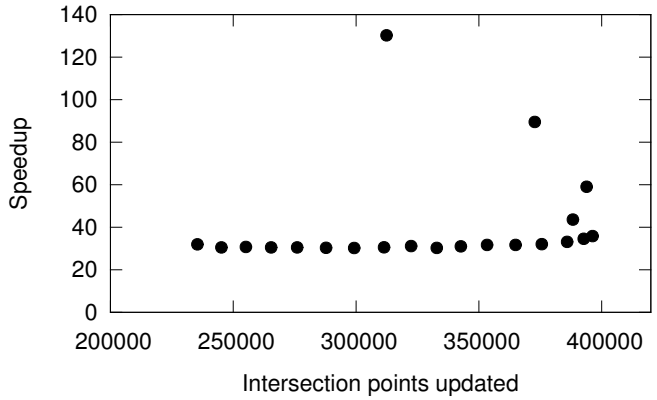
The results in Fig. 6.6 show speed-up factors on one of the more complex model pairs, shown in Figs. 6.5(a) and 6.5(b). The monkey model contains 1204 vertices, and the bird model has 275 vertices. Their reduced convolution contains 16,425 segments. When the monkey is scaled along the interval $s \in (0, \infty]$, there are 56,996,430 critical events.

Fig. 6.7 shows speed-up factors for the neuron$\widehat{\otimes}$disc pair shown in Fig. 6.1. There are 1815 vertices in the neuron model and 32 vertices in the disc model. The reduced convolution of the neuron and the disc has 3196 segments. There are 56238 critical events discovered when the disc is scaled across $s \in (0, \infty)$.

For more complex model pairs, the table shows that the update method outperforms recomputing the intersections by an order of magnitude in general. Of course, for larger scale jumps, the bottleneck of this method is in updating the events as P scales. This is



(a) Speed-up vs. Number of critical events processed



(b) Speed-up vs. Number of intersections updated

Figure 6.6: Speed-up as a function of critical events and number of intersections updated as P scales in 3D. The number of events processed is a much better predictor of the speed gain than the number of intersections updated, as events generally outstrip intersectins in number by orders of magnitude.

largely due to the high number of events that occur, especially in complex model pairs. In practice, however, scaling that needs to be done continuously tends to occur over smaller intervals that are generally quick to update.

The time to compute the scale bounds in 2D is on the order of the time necessary to compute the intersections using brute force. Asymptotically, the time bound of $O(n^2)$ for the initial computation to identify all intersections and all events is worse than the best-case bound for the static computation of intersections, $\Theta(n \log n)$ (with a constant number of

intersections). However, on average, the time taken to compute the events during dynamic scaling is a fraction of the time taken to compute the intersections. The monkey/bird model pair required on average about 140 seconds to build the initial intersections, but only about 28.7 seconds to compute the events. The neuron/circle pair took about 5.3 seconds to compute its nodes, but only 1.25 seconds to compute the events.

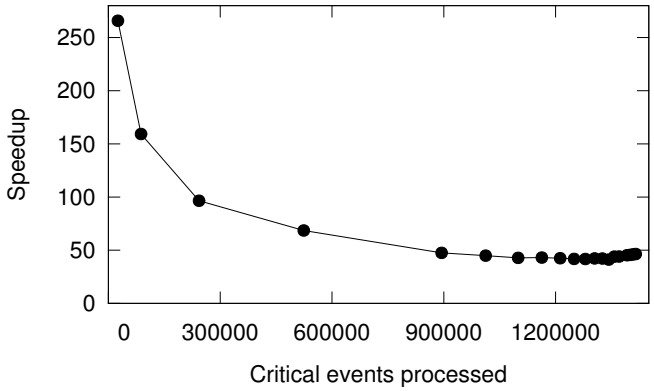
Even though both pre-processing steps are asymptotically similar, on average the time taken to compute the events is a fraction of the time taken to compute the intersections. The monkey/bird model pair required on average about 140 s to build the initial intersections, but only about 28.7 s to compute the events. The neuron/disc pair took about 5.3 s to compute its nodes, but only 1.25 s to compute the events.

With that in mind, given significant speed-up from our method only a small number of queries are needed to make up the difference in pre-computation time before it is vastly more expensive to perform the entire Minkowski sum computation again and again. Additionally, in this case it is reasonable to treat the initial computation of intersections as a pre-computation for situations requiring dynamic scaling queries. The advantage of the brute-force method then is its simplicity of implementation; there are fewer degeneracies to account for.

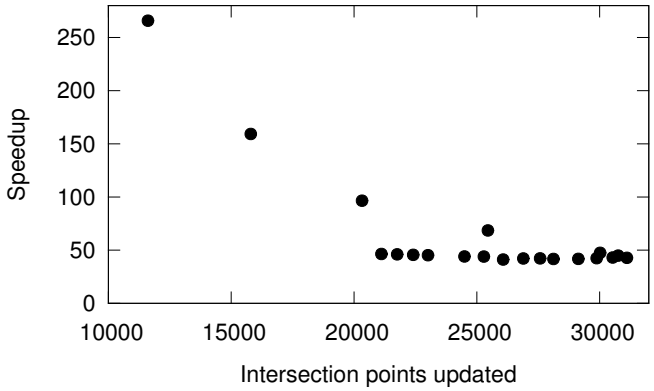
6.5.2 Results from Uniform and Non-Uniform Scaling in 3D

3D enumerative scaling under uniform scaling

The enumerative approach does not scale well in 3D. In Fig. 6.9, we see speed-up values for the $knot \oplus clutch$ model pair by number of events processed. Computing the event structure took approximately 65 minutes, and computed $8,427,881$ events on the scaling domain $[0.2, 2]$. We see a similar asymptotic behavior in speed-up over recomputing the line intersections from scratch, as we do in 2D. However, unlike 2D examples, the collision detection step that we use to filter out boundaries interior to the Minkowski sum boundary takes about 50% of the total computation time [40]. When we compute total overall speedup, the enumerative method does not provide a significant increase in speed, especially



(a) Speed-up vs. Number of critical events processed



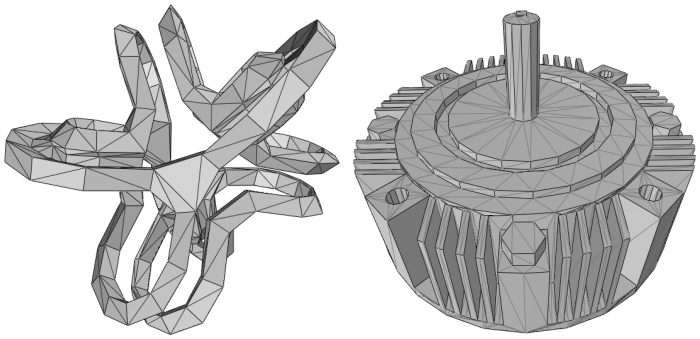
(b) Speed-up vs. Number of intersections updated

Figure 6.7: Speed-up factors for the neuron$\widehat{\otimes}$disc pair from Fig. 6.1. Again, number of events processed is a good predictor for speed-up, while number of intersections processed is not, though in this case it correlates more consistently than in monkey$\widehat{\otimes}$bird. (a) Speed-up by critical events for neuron$\widehat{\otimes}$disc. (b) Speed-up by intersections updated for neuron$\widehat{\otimes}$disc.

considering the extremely long time needed to compute the event structure.

Non-uniform scaling for convex polyhedra

We tested the 3D non-enumerative update-based method against computation from scratch over 100 random non-uniform scaling factors. Note that we have no experimental results for convex models in the previous sections because scaling the convex models uniformly simply makes their Minkowski sums scale uniformly. On the contrary, non-uniform scaling provides



(a) knot, 992 facets (b) clutch, 2116 facets

Figure 6.8: Knot and clutch models.

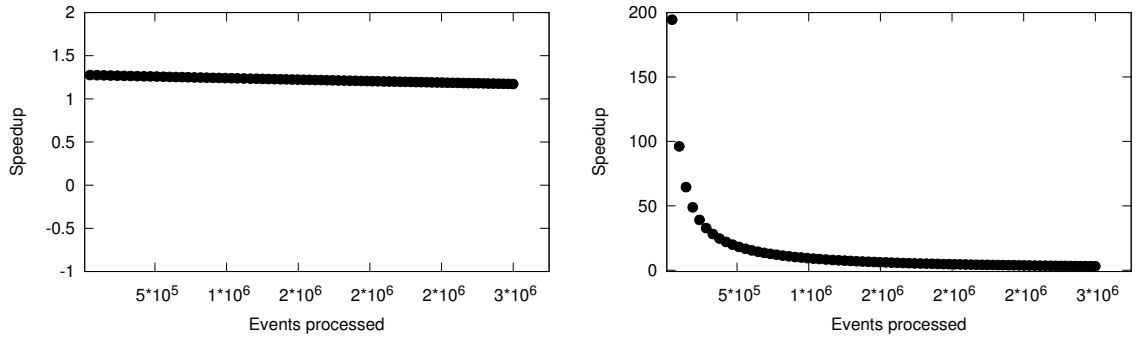


Figure 6.9: (a) Speed-up from updating $knot \oplus clutch$ using event enumeration, including time used for collision detection. (b) Speed-up for updating intersections only.

more complex and interesting behaviors to the Minkowski sums of convex models.

Our results for non-uniform scaling using models in Fig. 3.3 are shown in Table 6.2. The speedup values shown, computed again as $\frac{t_{static}}{t_{dynamic}}$, are the average speed-up over 100 queries. Here, we see significant speed-up (between 13 and 94) for computing convex Minkowksi sums dynamically over manual recomputation across all studied models. Note these figures are much less varied than the 2D case because the the convolution for convex pairs is always equal to the Minkowski sum boundary, and so no culling is necessary.

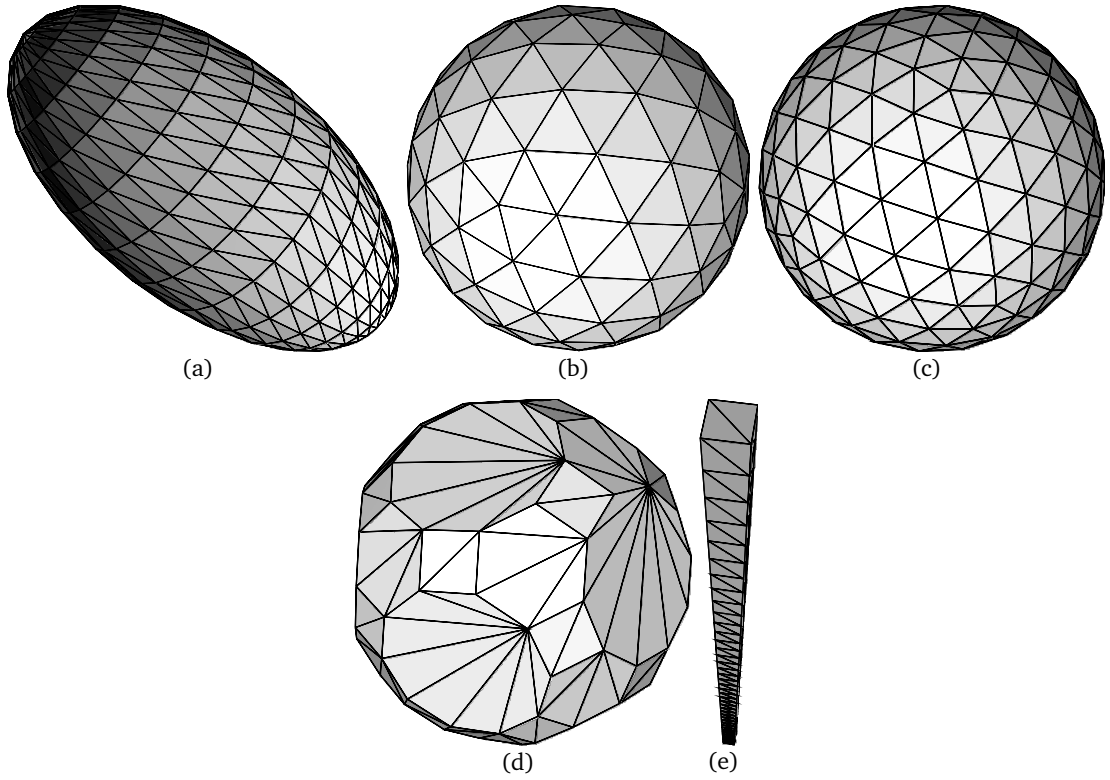


Figure 6.10: Models used in the experiments: (a) ellipse, 960 facets, (b) GS3, 320 facets, (c) GS4, 500 facets, (d) truncated icosidodecahedron (TI), 236 facets, (e) v-rod, 324 facets.

6.6 Applications

In this section, we discuss two direct applications of the proposed dynamic Minkowski sum method.

6.6.1 Motion planning and rapid virtual prototyping

Our method has applications to motion planning problems with repeated, scaled objects. As shown in Fig. 6.11, the example environment contains only 12 scaled objects yet has a speed-up factor of 38.8 over recomputing the intersections from scratch. Very few scaling requests are necessary to make up the difference caused by pre-processing the events. As a result, the efficiency of computing configuration spaces for motion planning environments is

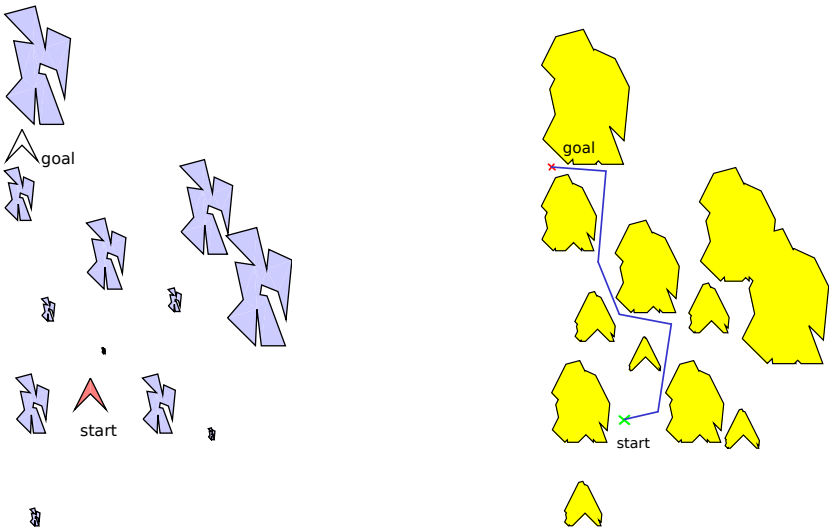
Table 6.2: Average speed-up of the 3D non-enumerative dynamic update algorithm over 100 random *non-uniform* rescalings. Models in the first column are the model being scaled in each experiment, while models in the first row are the static (unscaled) model. There is no collision detection component here, the values provided are total speed-up. The bolded values are, again, the lowest and highest speed-up values achieved.

	ellipse	$GS3$	$GS4$	TI	v-rod
ellipse	17.231	16.653	32.312	14.636	19.060
$GS3$	44.475	18.311	17.584	17.423	29.902
$GS4$	**13.109**	17.273	13.050	18.026	34.342
TI	18.685	20.332	19.669	24.952	37.960
v-rod	24.471	40.246	40.691	44.757	**97.410**

greatly enhanced.

Furthermore, the method can also be used for a robot of uncertain size to answer the following query: "Given a base assumed size for the robot, what is the largest scale factor for which a given path is valid?" by computing events for the robot's scaling. Similar to the contact point computation for intersection events, such events can be computed for segments of the robot's path. This allows reporting of the scale of first contact between an obstacle boundary and the robot's path. In fact, computing these events is easier than computing the convolution events, because the robot's path in this case does not vary with scale.

In rapid prototyping, it is important to minimize waste of both time and materials by guaranteeing that the parts of any assemblies produced will fit together once they are constructed from the CAD input. However physical objects are manufactured to tolerance, while CAD models do not have this limitation. The models can clip through each other, or produce physical output that is larger than the virtual model specifies. In a particular problem called part removal, which is similar to the motion planning application discussed above, one possible method of dealing with this uncertainty in size is to determine tight bounds on the physical sizes that will actually fit into the assembly without slipping out or causing unwanted motion. By first computing the upper bound on the scale of the part that will fit to the configuration in the assembly, we can then find the smallest scale for which the assembly traps it in place by specifying an escape path through the assembly and computing



(a) Example environment (b) Corresponding C-space

Figure 6.11: An example environment with repeated scaled obstacles. In (a), unfilled polygons are the obstacles, the filled polygon is the robot. Computing the convolution intersections for the example environment required 96.811 ms recomputing the intersections from scratch, but only 2.498 ms using our update method, a speed-up factor of approximately 38.755.

the largest valid part scale for which that path is valid.

6.6.2 Feature detection and shape decomposition

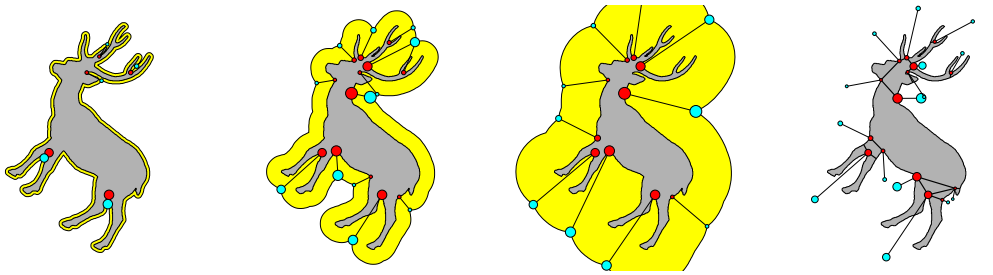
Minkowski sum involving a dynamically-scaling disc or sphere has strong connection to the medial axis (MA), an important shape descriptor. Follow the notation used in α-shapes by Edelsbrunner et al. [23], we call a disc and a sphere with diameter α, an α-disc and α-sphere.

Lu et al. [44] proposed a method to create shape decompositions by detecting the intersections of convolution between a polygon and a sequence of α-discs. They show that the vertices in the arrangement of convolution are closely related to well-known features, such as bridges and pockets, related to the concavity of polygons. More specifically, a **bridge** β of a given polygon P is a segment $\beta = \overline{vu}$ that lies completely in the space exterior to P, where v and u are two points on the boundary ∂P of P. More specifically, a segment $\overline{vu}$ is a

bridge of P if and only if $v, u \in \partial P$ and the open set of $\overline{vu}$ is a subset of the complement of P, and an α-**bridge** is a bridge between the tangent points of an empty α-disc centered at an intersection x of the convolution.

By varying the value of α, these α-bridges are born and die, thus the persistence (i.e., life span) of bridges provide an important measurement to the significance of concave features. More specifically, when $\alpha = 0$, the bridges correspond to the reflex vertices and when $\alpha = \infty$, the bridges correspond to the convex hull edges of the polygon. Interestingly, these intersections parameterized by α also implicitly trace out the Voronoi Complex (VC) in the space *exterior* to P. Thus, the persistence analysis corresponds to measuring the length of the segments on the VC.

In [44], the convolution and its intersections are computed from scratch when α changes. As shown in Appendix, there is a straightforward way to modify Section 6.2 to handle the situation with arcs. This approach allows us to continuously update the concave features for multiple α values at only critical events. The speed-up for the decomposition of the polygons shown in Fig. 6.12 using the proposed method is 68.556 over that of repetitive re-computation of the Minkowski sums.



(a) $\alpha = 0.05$ (b) $\alpha = 0.2$ (c) $\alpha = 0.4$ (d) $0.05 \leq \alpha \leq 1$

Figure 6.12: The lighter (blue) discs on the outer boundary are convolution intersections, and the darker (red) discs on the beetle polygon are pocket minima. Corresponding pairs are connected. Several concave features disappear after α increases from 0.05 to 0.2. Figure (d) shows the decompositions using persistence analysis for $0.05 \leq \alpha \leq 1$.

Chapter 7: Convex mappings of non-convex polyhedra and their applications

7.1 Introduction

In Chapter 5 we discussed dynamic methods for rotating convex polyhedra, but many inputs used in practical applications are non-convex. As we have noted, the time complexity for computing Minkowski sums of non-convex polyhedra can be much larger, by as much as four orders of magnitude. Therefore, if we can extend our methods to non-convex polyhedra, we can address a much more general set of inputs and solve a wider array of problems.

In considering how to extend our existing approaches to non-convex polyhedra, a natural question that arises is: does a convex representation of a non-convex input exist that is useful for our purposes? It turns out that the answer to this question is that such a representation does exist. The arrangement of a Gauss map for a given polyhedron is embedded on the surface of a unit sphere.

Although the sphere itself is a convex object, it does not provide the kind of representation we originally envisioned for methods like the dynamic Minkowski sum described in Chapter 5, which accepts only polyhedral meshes. However, it turns out that we can apply these same methods to the arrangement of the Gauss map with no significant modification.

Measuring interpenetration between two models in collision is an important problem in a number of applications, including geometric modeling, computer graphics, and algorithmic robotics. Penetration depth (PD), which is defined as a minimum translation to separate models in collision [17, 20], is a commonly used measure of interpenetration. In dynamics simulations, assembly planning, robotic motion planning, or six-degree-of-freedom haptic

rendering, collision happens frequently due to numerical errors, controls errors, discretization of motion, or interface latency. Collision states are often considered invalid in these applications, as materials in non-simulated environments often cannot interpenetrate in this way, and PD is often used in reverting such an invalid state to a valid, collision-free state; an approach known as a penalty-based system.

If P and Q overlap, the penetration depth $PD(P, Q)$ is defined as the minimum distance between the common origin of P and Q, $\mathbf{o}$ and the boundary surface of the Minkowski sums, $\partial(P \oplus -Q)$ [20]:

$$PD(P, Q) = \{\min \|\mathbf{q}\| \mid \mathbf{q} \in \partial(P \oplus -Q)\}. \tag{7.1}$$

In other words, we can compute PD between P and Q by projecting $\mathbf{o}$ onto the surface of the Minkowski sum $\partial(P \oplus -Q)$.

However, the definition of PD in Eq. 7.1 can lead to a discontinuity in the the gradient of the PD when $\mathbf{o}$ crosses the medial axis of $\partial(P \oplus -Q)$ [2]. This discontinuity is a significant problem in penalty-based systems [48] that rely on PD, as it can cause the simulation to become extremely unstable.

Recently, there has been some research into solving the discontinuity problem via a new PD approach known as continuous penetration depth (CPD). Zhang et al. [56] first proposed a method for computing CPD using spherical parameterization of configuration spaces. However this definition can be only be applied when the Minkowski sum of the inputs is homeomorphic to a sphere with no holes.

Lee and Kim [35] proposed another CPD approach using Phong projection [35]. Unfortunately both approaches are initial forays into this area of research, and they have limited use in practical applications, since they do not function when a model rotates or contains holes.

However, by using our approach of computing local Minkowski sums dynamically via the convex map [53], Lee et al. also recently proposed another approach, published alongside our work in [53], which not only provides the ability to efficiently estimate CPD for models under rotation and with holes, but also provides more optimal CPD estimation than previous

methods.

That paper defines a construction method of a convex polyhedron based on the arrangement of the Gauss map. It has since been shown that the construction defined in that paper is not correct–it is not capable of constructing all possible combinatorial structures of convex polyhedra. However, by the using the arrangement of the Gauss map directly instead of relying on the convex map, we are able to avoid these concerns.

We further introduce a modified method of the dynamic Minkowski sum originally shown in Chapter 5, which uses the arrangement of the Gauss map instead of the convex mapping, combine with gap-filling to patch the Minkowski sum. Finally, we discuss in more depth the main results achieved by using our new methods for continuous penetration depth estimation.

7.2 Dynamic convolution using the arrangement of the Gauss map

We can use the arrangement of the G to dynamically construct the convolution $P \otimes Q$ of two arbitrary polyhedra. First we observe that each point $G(\overline{f}) \in G(P)$ represents a (possibly empty) set S which is a subset of the faces in P which have the same outward normal direction as $\overline{f}$, we call this set $\overline{F}$ the *correspondence set* of $\overline{f}$. Similarly, each edge $\overline{e}$ and vertex $\overline{v}$ represents a possibly empty set which is a subset of the edges or vertices, respectively, of P. We denote these correspondence sets $\overline{E}$ and $\overline{V}$ respectively. The following things are true:

1. Given any region $G(\overline{v}) \in G(P)$, for every $v \in \overline{V}$, $G(\overline{v})$ is wholly contained within $G(v)$, that is, $\forall \overline{v} \in G(P)$, $\forall v \in \overline{V} : G(\overline{v}) \subset G(v)$.

2. Given any edge $\overline{e} \in G(P)$, for every $e \in \overline{E}$, $G(\overline{e})$ is wholly contained within $G(e)$ ($G(\overline{e}) \subset G(e)$).

3. Given any face $\overline{f} \in G(P)$, for every $f \in \overline{F}$, $G(\overline{f}) = G(f)$.

As a consequence, given $G(P)$ and $G(Q)$, we have that if $G(\overline{f}) \in G(P)$ is in $G(\overline{v}) \in Q$, then $\forall f \in \overline{F}, v \in \overline{V}$, f is compatible with v. Similarly, for intersecting $G(\overline{e}_P) \in G(P), G(\overline{e}_Q) \in G(Q)$, $\forall e_P \in \overline{E}_P, e_Q \in \overline{E}_Q$, we have that e_P is compatible with e_Q.

The direct correspondence between the labels of the Gauss map overlay of $G(P)$ and $G(Q)$ and $P \otimes Q$ enables us to use the techniques from Chapter 5 to update the convolution dynamically as Q rotates. First, we note that as Q rotates, so does $G(Q)$–this is a trivial observation since the outward normals of $f \in F_Q$ rotate the same as the outward normals of $f \in \overline{F}_Q$. Secondly, we observe that the degree of incompatibility described in Chapter 5 uses gradient descent to find corresponding compatible faces through the use of face normals. This does not require us to have the exact shape of the input; the Gauss map suffices for us to use the method directly. However, because the inputs are non-convex, this yields not a dynamic Minkowski sum but a dynamic convolution.

7.2.1 Constructing local Minkowski sums using the convex map

Given an ϵ-ball, S_i and two polyhedra, P and Q, we are interested in finding the local Minkowski sum boundary $\partial\mathcal{M} = \partial(P \oplus -Q) \cap S_i$ at *interactive rates*: a task that no traditional approaches is able to accomplish. In this section, we first present a robust but slower method that determines $\partial\mathcal{M}$ without considering temporal coherence and then the idea of *gap filling* is presented to exploit temporal coherence. In our implementation, the first method is used as a bootstrap step to populate S_i and also as the last step in filling the missing facets. This new approach can also be viewed as a hybrid method that takes the advantages from the decomposition-based and convolution-based approaches.

7.2.2 Bootstrapping using Bounding Sphere Hierarchy

Constructing the entire Minkowski-sum surface of two polyhedra is prohibitively slow for real time applications. Here we show that the computational efficiency can be significantly improved if the computation domain is confined within a small sphere S_i. The key idea is to

cull the computation that generates the facets far from S_i . Let $\mathcal{S}(X)$ be the bounding sphere hierarchy of an object X and S_X be a node in $\mathcal{S}(X)$. The following theorem provides the main mechanism in the culling procedure.

Theorem 2. *Let $P' = P \cap S_P$ be a nonempty subset enclosed by sphere S_P. Similarly, $Q' = Q \cap S_Q$ is a nonempty subset in sphere S_Q. If $(S_P \oplus S_Q) \cap S_i = \emptyset$, then it is guaranteed that the Minkowski sum $(P' \oplus Q') \cap S_i = \emptyset$. Alternatively, if $(S_P \oplus S_Q) \subset S_i$, then it is guaranteed that the Minkowski sum $(P' \oplus Q') \subset S_i$.*

Note that $S_P \oplus S_Q$ is simply a sphere with radius $r_P + r_Q$ centered at $\mathbf{c}_P + \mathbf{c}_Q$, where r_X and $\mathbf{c}_X$ are the radius and the center of the sphere X. Theorem 2 implies a culling procedure that starts from the a pair of root nodes of $\mathcal{S}(P)$ and $\mathcal{S}(Q)$. If $(S_P \oplus S_Q)$ is neither entirely inside nor outside S_i then all children pairs of S_P and S_Q are further tested.

The culling procedure can be further optimized near the bottom of the hierarchies. If S_P encloses a single triangle t_P of P, and S_Q encloses only t_Q, then it is not difficult to see that the smallest bounding sphere of $t_P \oplus t_Q$ can be much smaller than $S_P \oplus S_Q$, for example, when t_P and t_Q are long sliver triangles perpendicular to each other. Therefore, when the culling process reaches the bottoms of $\mathcal{S}(P)$ and $\mathcal{S}(Q)$, a sphere tightly enclosing $t_P \oplus t_Q$ is constructed and compared to S_i.

Given a pair of internal nodes S_P and S_Q, their enclosed subsets P' and Q' contain multiple facets and can be convoluted using their gauss maps $G(P)$ and $G(Q)$. We then gather the local convolution facets inside S_i, and these facets are subdivided and stitched into 2-manifold patches in a way such that each patch can be classified as either on the Minkowski-sum surface or in its interior using the method proposed in [41].

7.2.3 Gap filling and Error repair

Let $\partial\mathcal{M}_{i-1}$ denote the local Minkowski sum constructed in iteration $i-1$. As we pointed out in Chapter 5, the combinatorial structures of $\partial\mathcal{M}_{i-1}$ may become invalid after Q rotates and result in convolution errors and Minkowski-sum errors. In this section, we will present a

method that constructs $\partial\mathcal{M}_i$ for the iteration i by repairing these errors. Note that we do not make any assumptions regarding the amount of rotation applied to Q. In some sense, the computation time of our approach is sensitive to the amount to rotation. Larger amount of rotation between consecutive frames leads to bigger errors, and thus longer computation.

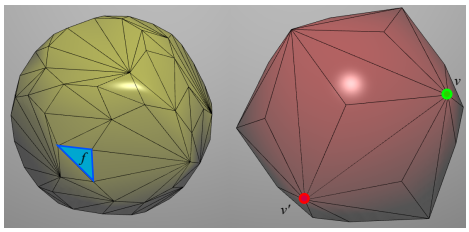


Figure 7.1: We identify and repair convolution errors. The (blue) face f on the left was compatible with the vertex v on the right before rotation. After rotation, our method finds f's new associated vertex v' using gradient decent on the Gauss maps.

Repairing Convolution Errors Removing convolution errors (i.e. incompatible features) results in gaps in the convolution surface. These gaps need to be filled with new compatible features. We find these features using the dynamic convolution described in Section 7.2. Both the convolution errors and the new compatible features can be identified using this method. We identify these errors using the method described in Section 7.2. Fig. 7.1 illustrates an example of the dynamic convolution repairs.

The repaired convolution is guaranteed to form 2-manifold patches inside S_i. However, because the repairs start from the remaining valid convolution facets, it is possible that the recovered facets are incomplete. Fig. 7.2 illustrates such an example. As we will discuss next, these missing facets may be discovered when we repair the Minkowski-sum errors.

Repairing Minkowski-sum Errors Given the repaired convolution surface, the surface again can be subdivided into 2-manifold patches and each patch can be classified as either

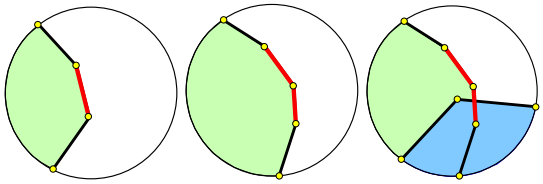


Figure 7.2: A red edge of the convolution (left) is identified as an error and repaired (middle). The ϵ-ball may contain facets that are not continuously connected to the existing facets. For example the region illustrated in blue (right).

on the Minkowski-sum surface or in the interior.

Observation 5. *A convolution patch classified as Minkowski-sum surface must form a 2-manifold whose boundaries must be on the surface of S_i. An example is shown in Fig. 7.3.*

If this property is violated, that means some convolution facets are missing and are not continuously connected to the existing convolution facets inside the ball (i.e. the example shown in Fig. 7.2). To identify these missing convolution facets, we let X be a 2-manifold convolution patch. Assume that there exists at least one point $x \in \partial X \setminus \partial S_i$. We construct a query ball $S(x)$ centered at x that is furthest away from ∂S_i and has radius equal to the distance between x and ∂S_i. Finally, the procedure described in Sec. 7.2.2 is invoked to construct the surfaces in $S(x)$.

Type	Full	Brute F.	BSH	Gap Fill.
Cone/Axes	58.91	16.10	24.23	10.33
Spoon/Cup	1393.61	453.29	69.38	13.91
Fish/Torus	1719.12	584.39	185.82	28.80
Torus/Torus	2842.93	1158.74	390.65	36.06

Table 7.1: Average Minkowski computation time in ms. The second to fourth columns represent the average running time of computing the full Minkowski sum, local Minkowski sums using brute force, bounding sphere hierarchy and the gap filling approaches. The radius of the ϵ-ball is 50. See Fig. 7.4.

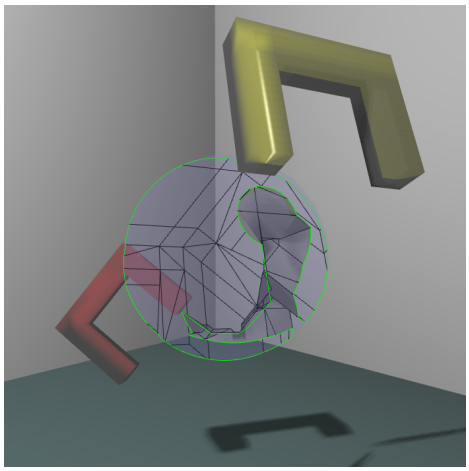


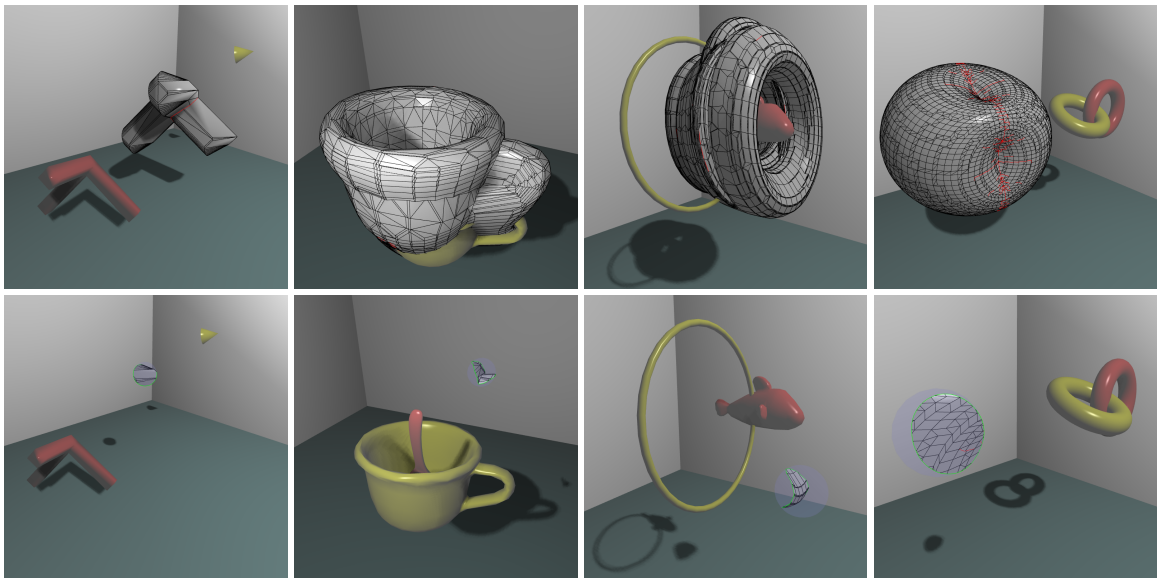
Figure 7.3: Minkowski sum of two U shapes. The boundaries of the Minkowski sum (colored in green) must be on the surface of S_i.

In Table 7.1, we compare the proposed dynamic Minkowski sum methods using bounding sphere hierarchies and gap filling to the method that computes the full Minkowski sum and a brute force method that computes all the convolution facets without using the dynamic convolution and uses only the ϵ-ball to filter the facets outside the ball. The radius of the local ϵ-ball used in these experiments is 50. The smallest bounding spheres of the full Minkowski sums of all examples have radii either equal to or less than 250 in all examples. Example outputs of the full and local dynamic Minkowski sums are shown in Fig. 7.4.

7.3 Continuous penetration depth estimation using local Minkowski sums

As mentioned above, Lee et al. [53] used our approaches to provide significant performance improvements to their continuous penetration estimates through the use of a Phong projection onto the Minkowski sum surface. We do not delve extensively into the details of their method here; see [53] for a deep discussion of their method and results. We will, however, discuss to some extent the results achieved by their method.

The use of dynamic convolutions, local Minkowski sums, and the gap-filling method,



(a) Cone/Axes (b) Spoon/Cup (c) Fish/Torus (d) Torus/Torus

Figure 7.4: Top: full Minkowski sum. Bottom: local Minkowski sum. The radius of the local ϵ-ball is 50. The smallest bounding spheres of the full Minkowski sums have radii equal to or smaller than 250 in all four examples.

allow a reduced computation time while still achieving the advantages of using the Minkowski sum surface to compute penetration depth. As noted in their results, the vast majority of their algorithm's time is spent computing the Minkowski sum–in many cases the Phong projection step takes less than 1ms. Therefore a significant amount of value is brought by the ability to compute dynamic Minkowski sums.

The difference in PD magnitude and smoothness also quite dramatic. The results obtained via the CPD algorithm are much smoother than those obtained via PolyDepth, and though not universally so, the CPD results are also often notably lower in magnitude, providing more optimal PD results as compared to PolyDepth, while still completing at interactive rates. For details on these results, please see the graphs in the results section of [53].

7.4 Analysis

We have shown that Gauss maps can be used to efficiently correct the convolutions of non-convex inputs, thus expanding on our work in Chapters 5 and 6. Based on this work, we have developed a method to compute local Minkowski sums using the convex map to identify and repair errors in the Minkowski sum boundary. We have also discuss a practical application of this new construction in the form of a continuous penetration depth estimation based on the construction of continuously updated local Minkowski sums, which demonstrates that our methods yield significant performance gains in solving a problem with wide applications to motion planning, computer design, haptics, and other geometric problems.

Chapter 8: Conclusions

The core work in this dissertation is the construction and presentation of new methods for building dynamic Minkowski sums by partial reuse of intermediate computation from an initial Minkowski sum. To that end, we have demonstrated a number of new methods for constructing such dynamic sums, using various strategies such as explicit computation of the dynamic space, partial or complete construction of the event space, and dynamically finding and repairing errors in the Minkowski sum surface. In addition, the contribution of the dynamic convolution and its application to repairing local Minkowski sums has already yielded a practical application in the form of a new, efficient continuous penetration depth algorithm.

8.1 Future Research

There remain many open problems in the space of dynamic Minkowski sum operations. In this section I discuss possible future research directions for dynamic Minkowski sum operations on general polyhedra.

8.2 Extensions of the dynamic convolution

Combined with the dynamic convolution algorithm presented in Chapter 7 Section 7.2, extensions of the methods laid out in Chapter 3 to 3D may be viable for producing complete dynamic Minkowski sums under rotation and non-uniform scaling for general polyhedra, rather than only convex polyhedra. By using the dynamic convolution to identify surface errors and rebuilding the convolution surface locally at error sites, it may be possible to

efficiently re-trim the convolution without running new collision detecting steps on many of the internal facets of the convolution set.

Remember that in Chapter 3 Section 3.3, we showed that boundary elements not participating in any closed loop, or that are part of a loop that does not define a properly-nested area are culled. The same idea functionally applies in three dimensions: faces in the convolution will not be on the Minkowski sum boundary unless they are part of a closed shell which defines a properly-nested volume inside the next outer-most boundary shell. (See Chapter 3 Section 3.3 for a refresher on what it means to be properly-nested.)

However it is entirely expected that in 3D, this approach still has the problem of yielding false shells akin to the false loops discussed in that method, which would need to be eliminated via collision detection. This is the expensive portion of the 2D algorithm defined in Chapter 3. However, in a dynamic rebuilding of a convolution, it may be possible to use prior information from the original trimming to identify when these culled faces do not need to be rechecked. This requires a more in-depth investigation into the properties and the temporal coherence of false shells than we can delve into in this section, but it bears investigation in future work.

8.3 Dynamic convolutions under generalized deformations of the input surfaces

A common geometric change that occurs during the design of objects–robots, part assemblies, and other such use cases–is the partial or total deformation of a mesh surface. We addressed a kind of total deformation in the form of model scaling of convex objects in Chapter 6, however local deformations are much more limited in scope, and can benefit from efficiency gains by only partially recomputing the Minkowski sum surface. Although at a passing glance, this may sound quite similar to the work we have done with local Minkowski sums for the continuous penetration depth application, it is actually quite different. In the work laid out in Chapter 7, local Minkowski sum patches are generated on a rigid model that has

undergone global transformation. This is not clearly extensible to a local change to a model resulting in a computation of a new, complete Minkowski sum.

We can imagine a limited set of primitive deformation operations that we would like to perform, under certain constraints such as the mesh being manifold and containing a well-defined volume:

- Translation of a connected component of model vertices
- Extrusion of a connected component of model faces
- Inverting convexity of edges between adjacent faces
- Insertion of a vertex on the surface of the model
- Deletion of a vertex on the surface of the model

These operations create a significant capability to deform the shape of the mesh in very localized ways. Although the translation of vertices creates errors in ways clearly analogous to the non-uniform scaling of convex polyhedra discussed in Chapter 6, the remaining operations may drastically alter the local area of the mesh in ways that do not map well to the existing notion of errors we have, and for which errors in the sense of the dynamic Minkowski sum methods discussed already do not provide a significant advantage, since we are making local patches whose locality is already quite clear to us upon making the update to begin with.

8.4 Minkowski sums under uncertainty

In environments whose exact geometries are not available in various applications, for example due to input uncertainty in the case of motion planning problems, or to bound uncertainty about the volume of the output when pieces are produced in situations of computer design and rapid prototyping applications, we may wish to compute Minkowski sums in such a way as to allow us to continue to act on the inputs we do have, and still

complete motion plans or assemblies that will be guaranteed to fit together. If we are given one input whose geometry is well-known, and a second input whose geometry is actually supplied as a set of possible variant geometries of the actual object to be represented. If we assume that these variants are, in a somewhat loose sense, similar to the actual object, then they should also be somewhat similar to each other, at least component-wise, and we may be able to compute Minkowski sum boundaries on these efficiently such that we can make guarantees about the computed spaces.

Although we may achieve the desired result by unioning the variant geometries together prior to taking a Minkowski sum, depending on the ways in which the inputs vary, the union may be much more complicated than the union of the Minkowski sums of its inputs. As a simplified example, if all inputs are convex but offset from each other, their union may yield a complex non-convex mesh for which it is much more expensive to compute the Minkowski sum.

In Chapter 3 we mentioned that Guerrero et al. [29] published a method for edit propagation based on our 2D Minkowski sum method which considers the concept of edit distance of two polygons. It may be possible to extend this work to polyhedra, in which case identifying non-similar patches of variant possible geometries of an uncertain true object may be simplified, and we may reuse the gap-filling approach to generate local patches from computation already done by computing the Minkowski sum of a random input from the set, and then correcting non-similar patches identified by such an algorithm.

Bibliography

[1] P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of Minkowski sums. In *European Symposium on Algorithms*, pages 20–31, 2000.

[2] D. Attali, J.-D. Boissonnat, and H. Edelsbrunner. Stability and computation of medial axes - a state-of-the-art report. In *Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration*, pages 109–125. Springer, 2009.

[3] F. Avnaim and J. Boissonnat. Polygon placement under translation and rotation. *STACS 88*, pages 322–333, 1988.

[4] C. Bajaj and T. K. Dey. Polygon nesting and robustness. *Inform. Process. Lett.*, 35:23–32, 1990.

[5] B. S. Baker, S. J. Fortune, and S. R. Mahaney. Polygon containment under translation. *J. Algorithms*, 7:532–548, 1986.

[6] E. H.-D. H. M. M. S. Baram, Alonand Fogel. *Exact Minkowski Sums of Polygons With Holes*, pages 71–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[7] H. Barki, F. Denis, and F. Dupont. Contributing vertices-based Minkowski sum of a nonconvex–convex pair of polyhedra. *ACM Trans. Graph.*, 30:3:1–3:16, feb 2011.

[8] J. Basch, L. J. Guibas, G. D. Ramkumar, and L. Ramshaw. Polyhedral tracings and their convolution. 1996.

[9] E. Behar and J.-M. Lien. A new method for mapping the configuration space obstacles of polygons. Technical Report GMU-CS-TR-2010-11, George Mason University, 2010.

[10] E. Behar and J.-M. Lien. Dynamic Minkowski sum of convex shapes. In *Proc. of IEEE Int. Conf. on Robotics and Automation*, Shanghai, China, May 2011.

[11] E. Behar and J.-M. Lien. Fast and robust 2d Minkowski sum using reduced convolution. In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, San Francisco, CA, Sep 2011.

[12] E. Behar and J.-M. Lien. Fast and robust 2d Minkowski sum using reduced convolution. In *Proc. IEEE Int. Conf. Intel. Rob. Syst. (IROS)*, San Francisco, CA, Sep. 2011.

[13] E. Behar and J.-M. Lien. Dynamic minkowski sums under scaling. *Computer-Aided Design*, Nov. 2012.

[14] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.

[15] R. Brost. Computing metric and topological properties of configuration-space obstacles. In *1989 IEEE International Conference on Robotics and Automation, 1989. Proceedings.*, pages 170–176, 1989.

[16] M. Caine. The design of shape interactions using motion constraints. In *1994 IEEE International Conference on Robotics and Automation, 1994. Proceedings.*, pages 366–371, 1994.

[17] S. Cameron and R. Culley. Determining the minimum translational distance between two convex polyhedra. In *Proceedings of International Conference on Robotics and Automation*, volume 3, pages 591–596, 1986.

[18] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, 1988.

[19] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[20] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri. Computing the intersection-depth of polyhedra. *Algorithmica*, 9:518–533, 1993.

[21] B. R. Donald. A search algorithm for motion planning with six degrees of freedom. *Artif. Intell.*, 31(3):295–353, 1987.

[22] H. Edelsbrunner. Lines in space: a collection of results. In *DIMACS Series Discrete. Math. Theoret. Computer Sci.*, volume 6, pages 77–93. 1991.

[23] H. Edelsbrunner, D. G. Kirkpatrick, and R. Seidel. On the shape of a set of points in the plane. *IEEE Trans. Inform. Theory*, IT-29:551–559, 1983.

[24] A. Elkeran. A new approach for sheet nesting problem using guided cuckoo search and pairwise clustering. *European Journal of Operational Research*, 231(3):757 – 769, 2013.

[25] A. Fabri, G. Giezeman, L. Kettner, S. Schirra, and S. Sch
”onherr. On the design of CGAL a computational geometry algorithms library. *Software Practice and Experience*, 30(11):1167–1202, 2000.

[26] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. In *Proc. 8th Wrkshp. Alg. Eng. Exper. (Alenex'06)*, pages 3–15, 2006.

[27] K. Fukuda. From the zonotope construction to the Minkowski addition of convex polytopes. *Journal of Symbolic Computation*, 38(4):1261–1272, 2004.

[28] P. K. Ghosh. A unified computational framework for Minkowski operations. *Computers and Graphics*, 17(4):357–378, 1993.

[29] P. Guerrero, S. Jeschke, M. Wimmer, and P. Wonka. Edit propagation using geometric relationship functions. *ACM Trans. Graph.*, 33(2):15:1–15:15, apr 2014.

[30] L. J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 100–111, 1983.

[31] D. Halperin, M. H. Overmars, and M. Sharir. Efficient motion planning for an L-shaped object. *SIAM J. Comput.*, 21(1):1–23, 1992.

[32] D. Halperin and M. Sharir. Arrangements and their applications in robotics: Recent developments. In *Proc. Workshop on Algorithmic Foundations of Robotics*, pages 495–511, 1995.

[33] F. Hoffmann, C. Icking, R. Klein, and K. Kriegel. The polygon exploration problem. *SIAM Journal on Computing*, 31(2):577–600, 2001.

[34] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 6th edition, 2006.

[35] Y. Lee and Y. Kim. Phongpd: Gradient-continuous penetration metric for polygonal models using phong projection. In *Proceedings of International Conference on Robotics and Automation*, pages 57–62, 2015.

[36] W. Li and S. McMains. A gpu-based voxelization approach to 3d Minkowski sum computation. In *SPM '10: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 31–40, New York, NY, USA, 2010. ACM.

[37] J.-M. Lien. Point-based Minkowski sum boundary. In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[38] J.-M. Lien. Covering Minkowski sum boundary using points with applications. *Computer Aided Geometric Design*, 25(8):652–666, 2008.

[39] J.-M. Lien. Minkowski sums of rotating convex polyhedra. In *Proc. 24th Annual ACM Symp. Computat. Geom. (SoCG)*, June 2008. Video Abstract.

[40] J.-M. Lien. A simple method for computing Minkowski sum boundary in 3d using collision detection. In *The Eighth International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, Guanajuato, Mexico, Dec 2008.

[41] J.-M. Lien. A simple method for computing minkowski sum boundary in 3d using collision detection. In *Algorithmic Foundation of Robotics VIII*, volume 57 of *Springer Tracts in Advanced Robotics*, pages 401–415. 2009.

[42] T. Lozano-Perez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, Feb. 1983.

[43] Y. Lu, E. Behar, S. Donnelly, J.-M. Lien, F. Camelli, and D. Wong. Fast and robust generation of city-scale seamless 3d urban models. In *SIAM Conference on Geometric and Physical Modeling (GD/SPM)*, Orlando, FA, Oct. 2011. SIAM/ACM.

[44] Y. Lu, J.-M. Lien, M. Ghosh, and N. M. Amato. α-decomposition of polygons. *Computer & Graphics*, SMI 12 special issue, 2012.

[45] N. Mayer, E. Fogel, and D. Halperin. Fast and robust retrieval of minkowski sums of rotating polytopes in 3-space. In *To appear in Proc. Symposium of Solid & Physical Modeling (SPM)*, 2010.

[46] S. Nelaturi and V. Shapiro. Configuration products in geometric modeling. In *SPM '09: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 247–258, 2009.

[47] J. O'Rourke. *Art Gallery Theorems and Algorithms*. The International Series of Monographs on Computer Science. Oxford University Press, New York, NY, 1987.

[48] M. Tang, D. Manocha, M. A. Otaduy, and R. Tong. Continuous penalty forces. *ACM Transactions on Graphics*, 31:107:1–107:9, July 2012.

[49] G. Varadhan and D. Manocha. Accurate Minkowski sum approximation of polyhedral models. *Graph. Models*, 68(4):343–355, 2006.

[50] R. Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *Proc. 14th Annual European Symposium on Algorithms*, pages 829–840, 2006.

[51] K. Wise and A. Bowyer. A survey of global configuration-space mapping techniques for a single robot in a static environment. *The International Journal of Robotics Research*, 19(8):762–779, 2000.

[52] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[53] J.-M. L. Youngeun Lee, Evan Behar and Y. J. Kim. Continuous penetration depth computation for rigid models using dynamic minkowski sums. In *Proc. Symposium on Physical & Solid Modeling (SPM 2016)*, Berlin, Germany, Jun 2016.

[54] H. Zhang, Pengand Zheng. *Generalized Contributing Vertices-Based Method for Minkowski Sum Outer-Face of Two Polygons*, pages 333–346. Springer International Publishing, Cham, 2015.

[55] L. Zhang, Y. J. Kim, G. Varadhan, and D. Manocha. Generalized penetration depth computation. *Comput. Aided Des.*, 39(8):625–638, 2007.

[56] X. Zhang, Y. J. Kim, and D. Manocha. Continuous penetration depth. *Computer-Aided Design*, 46:3–13, 2014.

Curriculum Vitae

Evan Behar graduated from Paul D. Schreiber High School, Port Washington, New York, in 2001. He received his Bachelor of Science in Computer Science and Mathematics from Rensselaer Polytechnic University in 2005. He was employed as a software engineer at SAIC for one year, IBM for two years, ComScore Inc. for one year, Google for two years, and Ellation, Inc. for one year. He is currently employed at Credit Karma. He received his Master of Science in Computer Science from George Mason University in 2010.