GPU-Based Computation of Voxelized Minkowski Sums with Applications

by

Wei Li

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Mechanical Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sara McMains, Chair
Professor David Dornfeld
Professor Carlo Séquin

Fall 2011

GPU-Based Computation of Voxelized Minkowski Sums with Applications

Abstract

GPU-Based Computation of Voxelized Minkowski Sums with Applications

by

Wei Li

Doctor of Philosophy in Mechanical Engineering

University of California, Berkeley

Professor Sara McMains, Chair

Minkowski sums are a fundamental operation for many applications in Computer-Aided Design and Manufacturing, such as solid modeling (offsetting and sweeping), collision detection, toolpath planning, assembly/disassembly planning, and penetration depth computation. Configuration spaces (C-spaces) are closely related to Minkowski sums; we analyze accessibility for waterjet cleaning processes as an example to illustrate the important relationship between them. We describe an algorithm for finding all the cleanable regions given the geometry of a workpiece. Minkowski sums are used to compute the C-spaces and cleanable regions are then found by visibility analysis.

Computing the Minkowski sum of two arbitrary polyhedra in $\mathbb{R}^3$ is difficult because of high combinatorial complexity. We present two algorithms for directly computing a voxelization of the Minkowski sum of two closed watertight polyhedra that run on the Graphics Processing Unit (GPU) and do not need to compute a complete boundary representation (B-rep).

For the first voxelization algorithm, we put forward a new formula that decomposes the Minkowski sum of two polyhedra into the union of the Minkowski sum of their boundaries and a translation of each input polyhedron. The union is then voxelized on the GPU using the stencil shadow volume technique. The performance of this algorithm depends on the numbers of faces of the two polyhedra.

For Minkowski sums in cases where we do not need to consider enclosed voids, we propose the second voxelization algorithm, which has much faster running times and also achieves higher resolution. It first robustly culls primitives that cannot contribute to the final boundary of the Minkowski sum, and then uses flood fill to find all the outer voxels. The performance of this algorithm depends on both the numbers of faces of the input polyhedra and the shape complexity of the Minkowski sum.

We demonstrate applications of the voxelized Minkowski sums in solid modeling, motion planning, and penetration depth computation. Compared with existing B-rep based algorithms, our voxelization algorithms are easy to implement and avoid the extra sampling process required in many applications.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere appreciation to my research advisor, Professor Sara Mc-Mains. She kindly accepted me as her student five years ago, when I decided to quit my job in China and pursue my Ph.D. degree in the U.S. She brought me into the interesting world of computer graphics, computational geometry, and GPU computing. Throughout the five years, her enthusiasm, attention to detail, and dedication to her research have greatly impressed me.

I would like to thank Professor Carlo Séquin and Professor David Dornfeld for being on my dissertation committee. Professor Séquin opened the door to the beautiful world of splines and surfaces for me while I took his solid modeling course in my first year. I also owe special thanks to Professor Alice Agogino, Professor Tarek Zohdi, and Professor Paul Wright who were on my qualifying exam committee and provided insightful suggestions on my research proposal. I would also like to thank Professor Jonathan Shewchuk for introducing me to floating point error analysis.

I thank Professor Jyh-Ming Lien from George Mason University for sending us his Minkowski sum codes for performance comparison and providing us with his test models, and Professor Vadim Shapiro for interesting discussions on Minkowski sums during his sabbatical in Berkeley. I also thank Professor Herbert Voelcker from Cornell University for scanning and sending us his technical report on Minkowski sums.

I am grateful to my lab colleagues, Youngung Shon, Rahul Khardekar, Adarsh Krishnamurthy, Xiaorui Chen, Yusuke Yasui, and Sushrut Pavanaskar for their helpful discussions and friendship. They have been great to work with. I would like to thank Saurabh Garg for discussing the problem and designing the test cases for the course project that led to chapter 3. I also thank Miguel Ávila for discussions on cleanability and his research on the cleaning effect of high-pressure waterjets.

My research in Berkeley has been supported by NSF and UC Discovery.

Last but not the least, I cannot find words to express my gratitude towards my parents and my wife Lin, for their unselfish love and their patience. Without them, none of my achievements would have been possible.

Chapter 1

Introduction

Minkowski sums are a fundamental operation for many applications in Computer-Aided Design and Manufacturing, such as solid modeling (offsetting and sweeping), collision detection, toolpath planning, assembly/disassembly planning, and penetration depth computation. The concept of configuration spaces (C-spaces), which have been heavily used in automatic toolpath planning, are closely related to Minkowski sums. In this thesis, we will analyze accessibility for waterjet cleaning (where mechanical components are cleaned using high pressure waterjets) as an example to illustrate the important relationship between them.

Despite the simplicity of its mathematical definition, computing the Minkowski sum of two arbitrary polyhedra in $\mathbb{R}^3$ is generally difficult because of its high combinatorial complexity. In this dissertation, we present two algorithms for directly computing a voxelization of the Minkowski sum of two closed watertight polyhedra. Unlike most previous algorithms for computing Minkowski sums, these two algorithms run on the Graphics Processing Unit (GPU) and directly create a voxelization without having to compute a complete boundary representation (B-rep).

1.1 Notation

In this dissertation we will not distinguish between points and vectors; i.e., a point will also represent the vector pointing from the origin to itself. Unless otherwise specified, we will use lower case letters for points and upper case for sets of points. Below is some notation frequently used in this dissertation.

O:	the origin
A_d:	the translation of point set A by a vector d
A^c:	the complement of point set A
$-A$:	A reflected about the origin (formally, $-A = \{-a \mid a \in A\}$)
∂A:	the boundary of an object A

1.2 Minkowski Sums

The Minkowski sum of two point sets A and B in $\mathbb{R}^n$ is defined as

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \tag{1.1}$$

where a and b denote the coordinate vectors of arbitrary points in A and B, and $+$ denotes vector addition. From the above definition, and the commutativity and associativity of vector addition, the following two properties (commutativity and associativity) of Minkowski sums hold:

$$A \oplus B = B \oplus A, \tag{1.2}$$

$$(A \oplus B) \oplus C = A \oplus (B \oplus C)\,. \tag{1.3}$$

The translation of a point set A by a vector d, denoted as A_d, can be represented as the Minkowski sum of point set A and another point set that contains only point d; i.e.,

$$A_d = A \oplus \{d\}\,. \tag{1.4}$$

In this dissertation, we will always denote $A \oplus \{d\}$ as either $A + d$ or A_d for simplicity. The shape of a Minkowski sum is translation invariant. If we translate one of the input models by a specific vector d, the Minkowski sum will also translate by the same vector, or mathematically,

$$A_d \oplus B = A \oplus B_d = (A \oplus B)_d\,. \tag{1.5}$$

Another equivalent definition of Minkowski sums using the concept of translation is

$$A \oplus B = \bigcup_{a \in A} B_a = \bigcup_{b \in B} A_b. \tag{1.6}$$

The equivalence of (1.1) and (1.6) can be shown as below:

$$\begin{array}{rl} & A \oplus B = \{a + b \mid a \in A, b \in B\} \\ \Longleftrightarrow & A \oplus B = \{a + B \mid a \in A\} \\ \Longleftrightarrow & A \oplus B = \bigcup_{a \in A} B_a. \end{array}$$

Definition (1.6) implies a method often used to construct and visualize Minkowski sums. If A and B represent polygons in $\mathbb{R}^2$ or polyhedra in $\mathbb{R}^3$, $A \oplus B$ can be generated by "sweeping" A along the boundary of B and then taking the union of the sweep and B (or vice versa). Figure 1.1 shows the 2D Minkowski sum of a square and a triangle generated by sweeping the triangle along the boundary of the square. A 3D example is shown in Figure 1.2, where the 3D Minkowski sum is generated by sweeping the tetrahedron over the six boundary faces of the cube. The two examples given in Figure 1.1 and 1.2 are simple convex models. When both input models become non-convex, their Minkowski sum can still be generated by sweeping one over the other, but the final shape will become much more complex (see

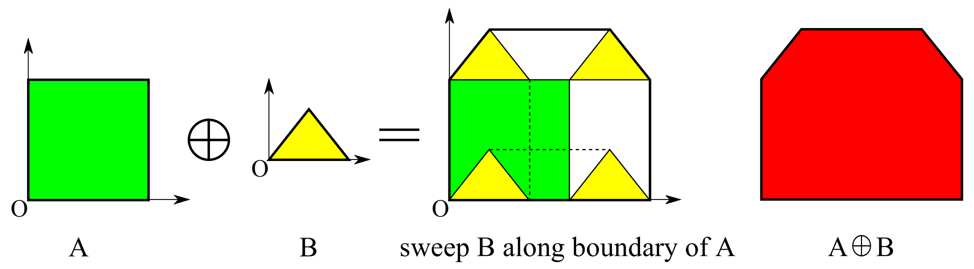


Figure 1.1: 2D Minkowski sum of a square and a triangle.

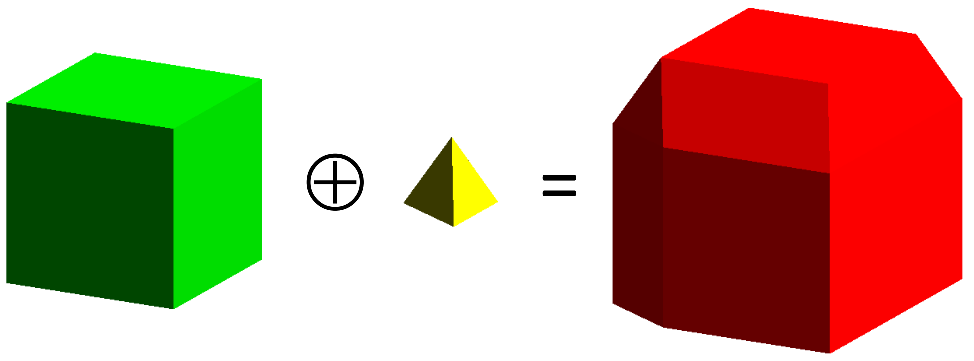


Figure 1.2: 3D Minkowski sum of a cube and a tetrahedron.

Figure 1.3 for an example). We will give a more formal and more accurate description of how Minkowski sums can be generated by such sweeps in Chapter 4.

Note that in Figure 1.1, we set the origin on the boundary of the input models. The position of the origin does not affect the shape of the Minkowski sum. From property (1.5), we know that changing the origin will only translate Minkowski sums without changing their shapes. In the following examples we will ignore the position of the origin unless position matters.

1.3 Applications of Minkowski Sums

Minkowski sums are a fundamental operation for many applications such as solid modeling, motion planning, collision detection, penetration depth computation, and mathematical morphology [Lozano-Pérez, 1983, Kaul and Rossignac, 1992, Varadhan et al., 2006, Nelaturi and Shapiro, 2009]. In this section we will discuss some example applications and illustrate how they can be solved by using Minkowski sums. More detailed theoretical background related to these applications will be discussed in section 2.1.

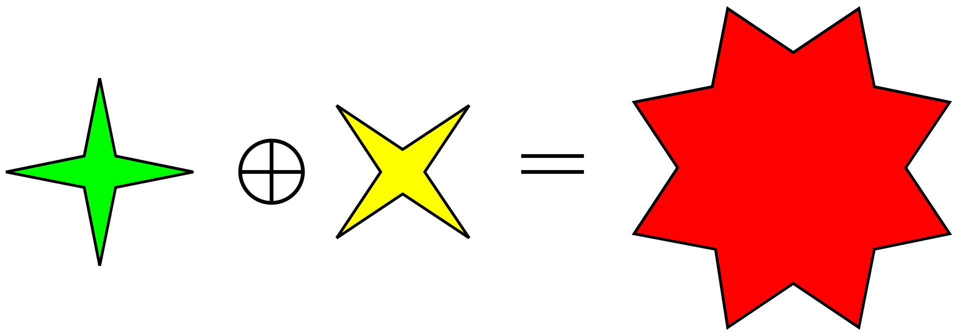


Figure 1.3: 2D Minkowski sum of two nonconvex geometries.

1.3.1 Solid Modeling

Minkowski sums can be used to compute offsets of geometric models. The outer offset of an object A with a distance r can be generated as $A \oplus Ball(r)$, where $Ball(r)$ is a ball centered at the origin with radius r. As long as the input model A is a closed watertight object, $A \oplus Ball(r)$ will always generate a closed watertight object. Figure 1.4 shows the outer offset of a table computed as its Minkowski sum with a ball.

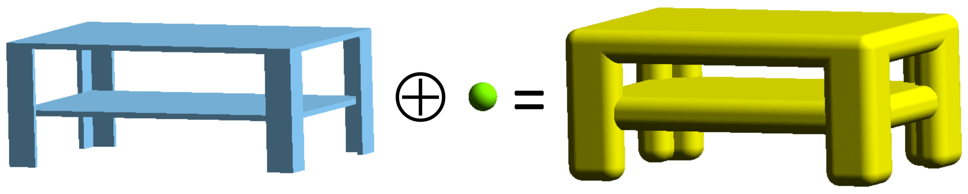


Figure 1.4: The outer offset of a table model.

The inner offset of A with a distance r is given by $(A^c \oplus Ball(r))^c$ where A^c denotes the complement set of A. A 2D example of inner offset computed using Minkowski sums is shown in Figure 1.5.

Minkowski sums can also be used to compute translation-only sweeps (the object to be swept does not rotate along the trajectory). Suppose the object to be swept is A and the trajectory is T, then the sweep of A along T can be expressed as $sweep(A,T) = \bigcup_{t \in T} A_t = A \oplus T$. An example of a sweep computed by Minkowski sums is shown in Figure 1.6.

1.3.2 Motion Planning

Minkowski sum based motion planners usually involve computing *configuration spaces* (C-spaces), introduced by Lozano-Pérez for motion planning of a rigid object among physical obstacles [Lozano-Pérez, 1983]. Every point in the C-space corresponds to a set of independent parameters that characterize the position and orientation of the rigid object (its

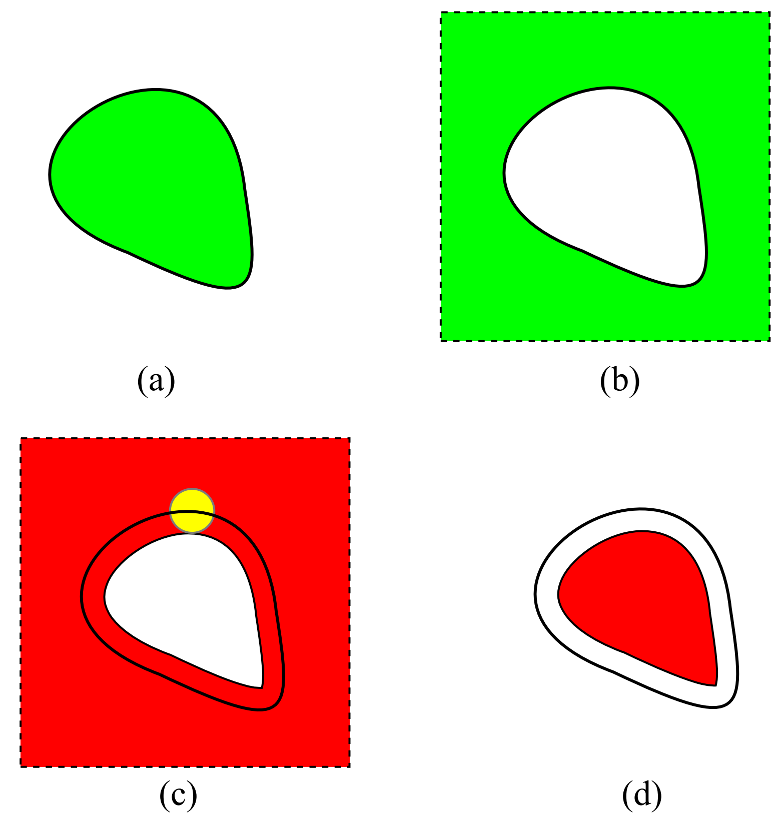


Figure 1.5: The inner offset of a 2D shape. (a) The 2D shape A. (b) The complement of A (A^c). (c) $A^c \oplus Ball(r)$. (d) The inner offset as $(A^c \oplus Ball(r))^c$.

configuration). In this section we assume fixed orientation, so the configuration is determined by the position. The whole C-space is usually decomposed into two sub-spaces, the *interference C-space* and the *free C-space*. The interference C-space is the set of configurations of the object where it collides with one or more of the obstacles, while free C-space is the set of configurations where it does not collide. The motion planning problem is then reduced to computing the interference and free C-spaces and finding a path in the free C-space connecting the initial and goal configurations.

Computation of C-spaces is based on the following important property of Minkowski sums: if A and B intersect each other, then the origin O must lie in set $A \oplus -B$, where $-B$ is set B reflected about the origin (formally, $-B = \{-b \mid b \in B\}$). The above property also forms the theoretical basis of almost all Minkowski sum based motion planning, collision detection, and assembly/disassembly algorithms. Figure 1.7 shows a 2D illustration. A formal proposition is given in section 2.1 (see Proposition 2.2). Note that the origin O can be chosen arbitrarily, since changing the origin O is equivalent to applying the same translation to both sets A and B, so it does not change either the position or the shape of

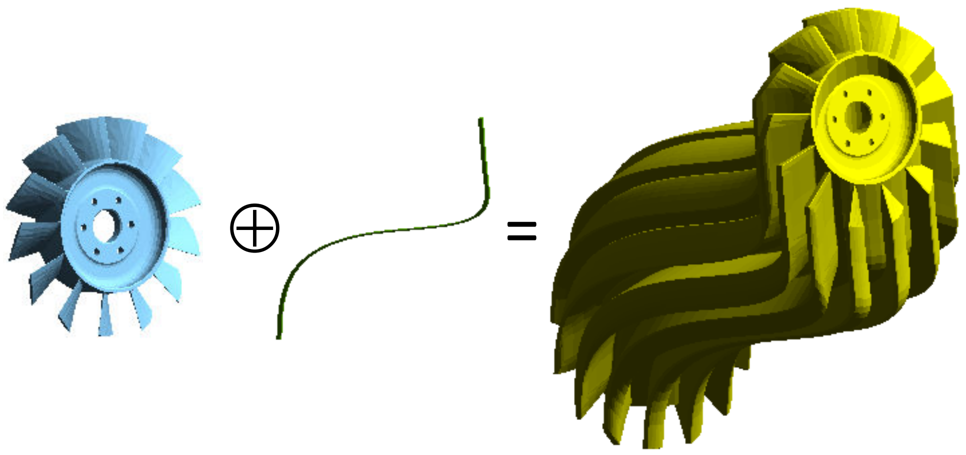


Figure 1.6: A 3D sweep computed by Minkowski sums.

$A \oplus -B$.

The interference and free C-spaces are usually computed using Minkowski sums based on Proposition 2.2. For P a translating object and Q the union of all the obstacles (see Figure 1.8, $Q = Q_1 \cup Q_2$), the interference C-space is $Q \oplus -P$. Sometimes $Q \oplus -P$ is also called the *C-space obstacle(s)*. In the configuration space, the obstacles are enlarged by the Minkowski sum operations and the moving object is reduced to a single point (initially it is located at the origin). Thus the collision detection problem between 3D objects is reduced to the simpler collision detection problem between a single point and a set of 3D objects. The free C-space is the complement set of the interference C-space. If we denote the interference and free C-spaces as S_i and S_f, then we have

$$\begin{aligned} S_i &= Q \oplus -P \\ S_f &= (Q \oplus -P)^c \,. \end{aligned} \tag{1.7}$$

Usually the moving object is constrained inside a workspace, denoted as W. To take the workspace into consideration, we can treat the complement of the workspace (W^c) as an obstacle, such that the object should not “touch” (i.e., collide with) the outside of the workspace. Then the formulas for S_i and S_f become

$$\begin{aligned} S_i &= (Q \cup W^c) \oplus -P \\ S_f &= S_i^c. \end{aligned} \tag{1.8}$$

If we let $Q = \emptyset$, the collision detection problem becomes an object containment problem (finding all the configurations under which object P is completely contained in the workspace W), and the formulas for S_i and S_f in this case become

$$\begin{aligned} S_i &= W^c \oplus -P \\ S_f &= S_i^c = (W^c \oplus -P)^c. \end{aligned} \tag{1.9}$$

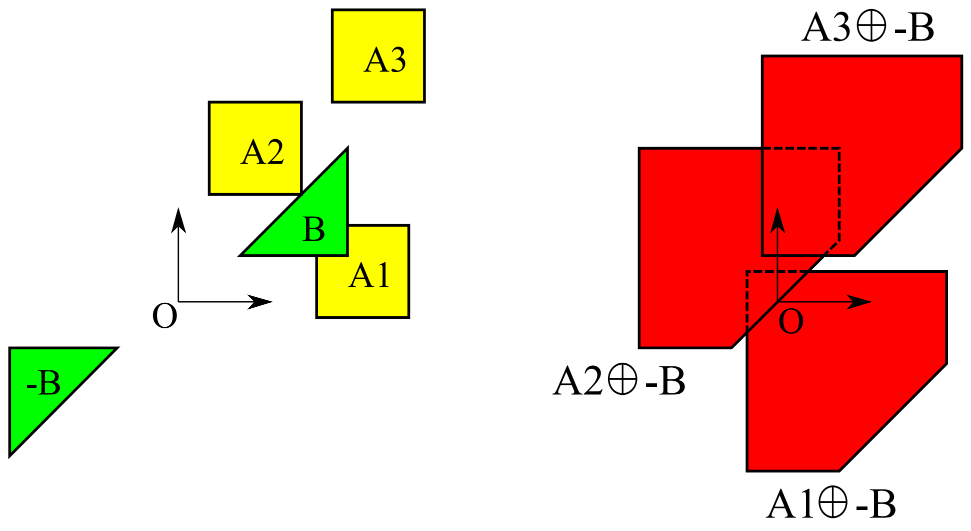


Figure 1.7: Collision detection by using Minkowski sums. A_1 intersects with B, A_2 just touches B on the boundary, and A_3 is separated from B. Correspondingly, O is in the interior of $A_1 \oplus -B$, on the boundary of $A_2 \oplus -B$, and outside of $A_3 \oplus -B$.

As opposed to $A \oplus B$, which means "enlarging" A by B, operation $(A^c \oplus -B)^c$ can be seen as a "shrinking" of A by B. When we computed the inner offset of A in section 1.3.1, we were actually using a special case of this expression, $(A^c \oplus \mathit{Ball}(r))^c$ (note that $\mathit{Ball}(r) = -\mathit{Ball}(r)$ for a ball centered at the origin). A more detailed discussion of $(A^c \oplus -B)^c$ can be found in Proposition 2.3.

A 2D example illustrating the basic process of C-space based motion planning is shown in Figure 1.8. In Chapter 3, we will discuss accessibility for waterjet cleaning in detail, which involves computation of several different C-spaces using Minkowski sums.

1.3.3 Penetration Depth Computation

Minkowski sums can also be used to compute the translational penetration depth between two intersecting objects. This penetration depth is the minimum translational distance to separate two intersecting objects. Mathematically, the penetration depth of two objects A and B is defined as:

$$\mathit{PenetrationDepth}(A, B) = \inf \left\{ |d| : d \in \mathbb{R}^3, A_d \cap B = \emptyset \right\}. \tag{1.10}$$

Penetration depth is often used in dynamic simulation, haptic rendering, and tolerance verification of CAD models. From section 1.3.2 we already know that if A and B intersect, then the origin O is inside $B \oplus -A$, but $B \oplus -A$ also reveals how "deeply" A and B intersect: the penetration depth is equal to the shortest distance from O to the boundary of $B \oplus -A$.

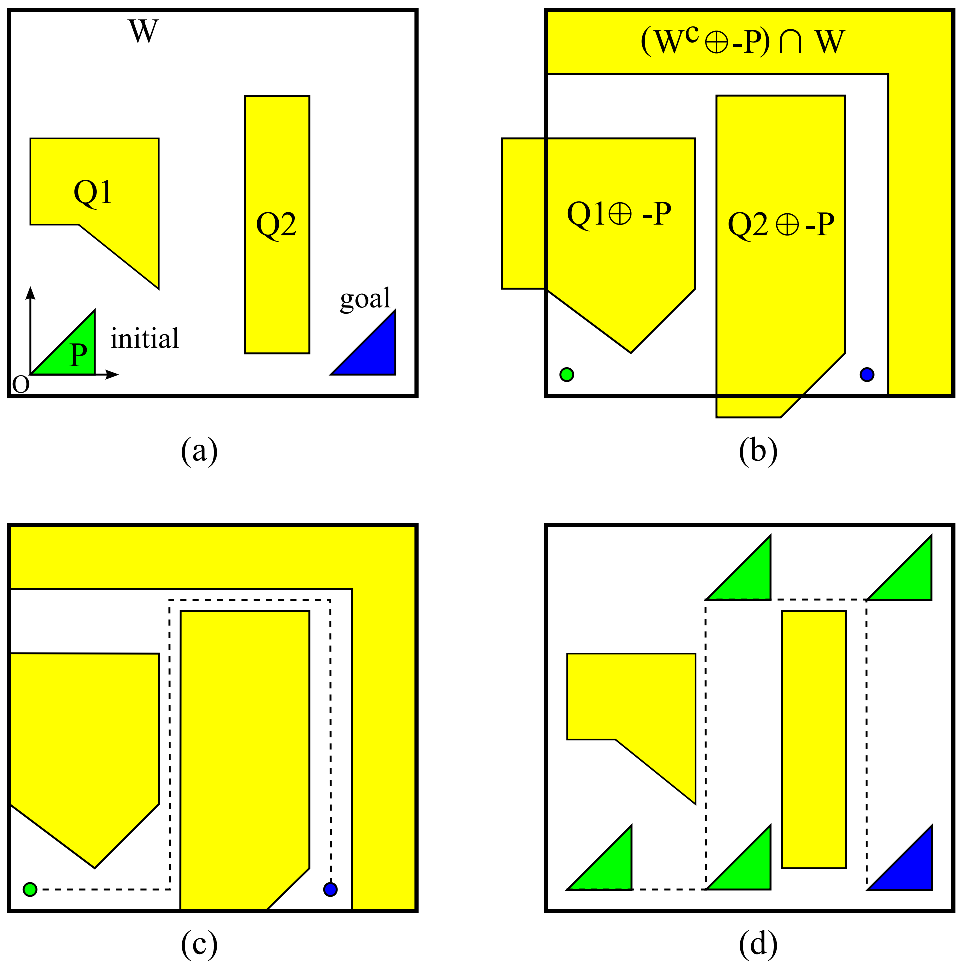


Figure 1.8: C-space based motion planning. (a) A moving object P and two obstacles Q_1 and Q_2 in a workspace W. We want to move P from the initial to the goal position. (b) C-space obstacles of Q_1 and Q_2 due to P. (c) A path contained in the free C-space connecting the initial and goal configuration. (d) The movement of the object in the workspace.

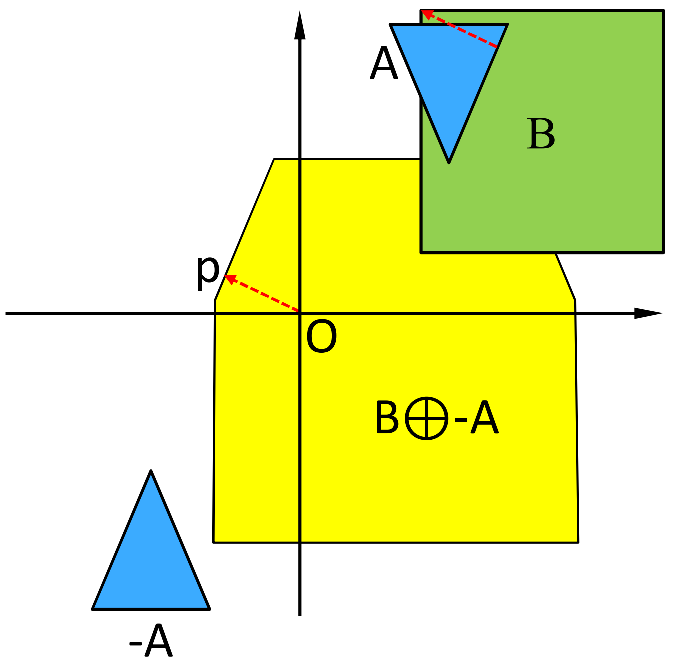


Figure 1.9: Penetration depth of A and B. The dashed red vector indicates the direction and distance in which to translate A to separate it from B.

The vector from O to the closest point on the boundary of $B \oplus -A$ gives the separation direction in which A can be translated away from B (see Figure 1.9).

1.4 Motivation

Despite the simplicity of its mathematical definition, computing the Minkowski sum of arbitrary polyhedra in $\mathbb{R}^3$ is generally difficult because of its high combinatorial complexity. For polyhedra A and B consisting of m and n facets respectively, although $A \oplus B$ only has complexity of $O(mn)$ if they are convex, the complexity can be as high as $O(m^3n^3)$ if they are non-convex [Varadhan and Manocha, 2006]. Most existing algorithms for computing general 3D Minkowski sums involve many complex 3D computations and their performance degrades rapidly as the polyhedra complexity increases (see section 2.3 for detailed discussions of existing algorithms). The two publicly available libraries, the CGAL library [CGAL, 2008] and "m+3d" provided by Lien [Lien, 2011], take minutes or even tens of minutes to compute Minkowski sums of models with just hundreds or thousands of triangles.

Most existing algorithms compute either an exact [Hachenberger, 2009, Fogel and Halperin, 2007, Halperin, 2002, Wein, 2006] or an approximated [Peternell and Steiner, 2007, Varadhan and Manocha, 2006] boundary representation (B-rep) of the Minkowski sum. A B-rep is convenient for rendering and visualization due to its explicit representation of surface boundaries, but it cannot be directly applied to many applications where we are more interested in the interior of the Minkowski sum instead of its boundary. In motion planning (see Figure 1.8 for an example), even if we have computed a B-rep based free C-space, it usually needs to be sampled in order to construct a connectivity roadmap [Varadhan et al., 2006, Lien, 2008c]. In collision detection and assembly/disassembly, we need to determine whether a point is inside or outside of the Minkowski sum, which usually requires a parity check for B-reps. If we consider higher dimensional Minkowski sums (e.g., 3D motion planning involving both translation and rotation requires computing and representing 6D Minkowski sums), a B-rep is not an appropriate representation any more.

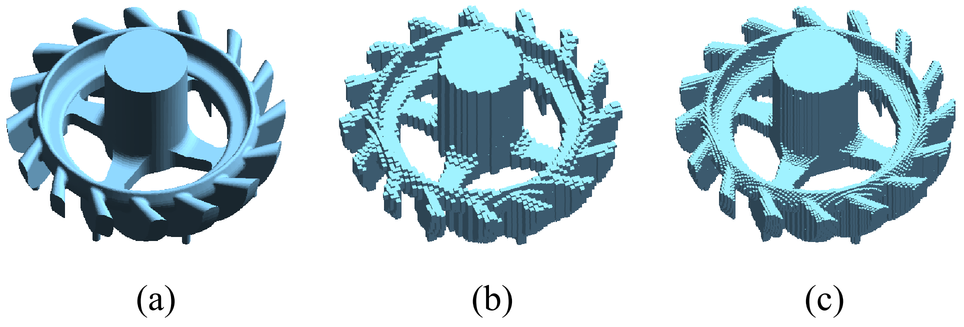


Figure 1.10: Representation of a 3D object. (a) The original B-rep model. (b) Voxelization of $64 \times 64 \times 64$. (c) Voxelization of $128 \times 128 \times 128$.

To overcome the shortcomings of B-rep based Minkowski sums, in this dissertation we consider a voxelized representation as an alternative to the traditional B-rep. A voxelized representation provides a uniform and simple description for objects and can be easily extended to higher dimensions. Figure 1.10 shows the voxelized representations of a 3D model at different resolutions. The voxelized representation of Minkowski sums is more advantageous in applications where a B-rep would need to be sampled or point membership classification (PMC, determining whether a point is inside a B-rep solid or not) would need to be performed. It provides sample points used to build the connectivity roadmap in motion planning with no need of further computation, and also provides immediate collision feedback by simply checking if a certain voxel is set to one or zero.

All the above factors have motivated us to research new approaches for directly computing a voxelized representation of Minkowski sums of arbitrary polyhedra, without having to compute a complete boundary representation. For achieving better performance than existing algorithms, we exploit the parallel computing capacity of modern GPUs. The built-in rasterization functionality of GPUs can also be utilized to compute the volumetric data.

1.5 Contributions

In chapter 3, we analyze the accessibility problem for waterjet cleaning as an example to illustrate the important relationship between C-spaces and Minkowski sums. We describe an algorithm for finding all the cleanable regions given the geometry of a workpiece. Implementations and results for 2D examples are shown to validate the approach.

In chapter 4, we mathematically describe and prove the commonly used "sweeping-along-the-boundary" method to generate Minkowski sums (Figure 1.1). Based on this method, we introduce a new formula that decomposes the Minkowski sum of two polyhedra into the union of the Minkowski sum of their boundaries and a translation of each input polyhedron. We also describe a method to voxelize the union on the GPU using the stencil shadow volume technique.

In chapter 5, we propose a new voxelization algorithm for computing the outer boundary of a Minkowski sum in cases where we do not need to consider enclosed voids, which extends upon previous work on Minkowski sums and combines these methods with GPU-based voxelization techniques. It runs completely on the GPU and is at least one order of magnitude faster than existing B-rep based algorithms.

Finally, in chapter 6, we demonstrate applications of the voxelized Minkowski sums in solid modeling, motion planning, and penetration depth computation.

Chapter 2

Background and Previous Work

In this chapter, we develop the theoretical background of some important Minkowski sum applications. We also discuss some commonly used approaches and existing algorithms for Minkowski sum computation and GPU-based voxelization.

2.1 Theoretical Background for Applications

In this section, we prove some mathematical propositions that form the theoretical basis of the applications described in section 1.3. These propositions are widely used in Minkowski sum publications, but to our knowledge have previously only been presented without formal proofs.

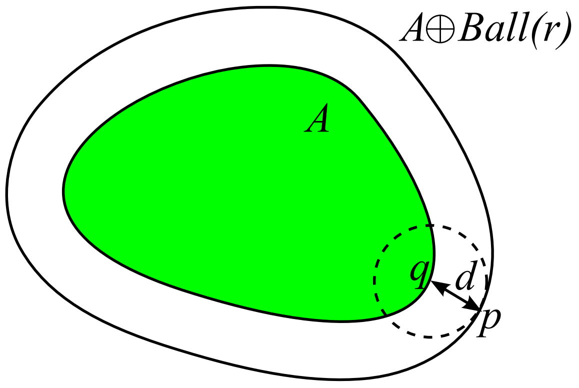


Figure 2.1: A point p on $\partial\,(A \oplus \mathit{Ball}\,(r))$ and its closest point q on A.

The first proposition relates to the offset application described in section 1.3.1. Again $\mathit{Ball}(r)$ denotes a ball centered at the origin with radius r; please refer back to section 1.1 for additional notation conventions.

Proposition 2.1. *Suppose A is a closed watertight object. Every point on $\partial(A \oplus \mathrm{Ball}(r))$ has a minimum distance of r to A.*

Proof. For any point $p \in \partial(A \oplus Ball(r))$, let its minimum distance to A be d, and its closest point (or one of them) on ∂A be q (see Figure 2.1). If $d < r$, point p is in the interior of $Ball(r) + q$, which contradicts the fact that $p \in \partial(A \oplus Ball(r))$. Then $d \geq r$.

On the other hand, suppose $p = a + b$, where $a \in A$ and $b \in Ball(r)$. If b is in the interior of $Ball(r)$, the infinitesimal ball around b is also in the interior of $Ball(r)$. Then the infinitesimal ball around $a+b$ is in the interior of $A \oplus Ball(r)$. This contradicts the fact that $p \in \partial(A \oplus Ball(r))$. Then $b \in \partial(Ball(r))$; i.e., b is a vector of length r. Then the distance between points p and a is r. Since d is the minimum distance from p to A and $a \in A$, we have $d \leq r$. This proves $d = r$. □

The next proposition is the basis for motion planning and collision detection algorithms (see the motion planning application in section 1.3.2).

Proposition 2.2. *Suppose A and B are closed watertight objects. Then*
(a) A and B intersect (including just contact) each other $\iff O \in A \oplus -B$;
or equivalently,
(b) A and B are separated $\iff O \notin A \oplus -B$.

Proof. We only prove (a), since (a) and (b) are equivalent.

$$\begin{aligned} A \text{ and } B \text{ intersect each other} &\iff A \cap B \neq \emptyset \\ &\iff \exists a \in A, b \in B, \text{ s.t. } a = b, \text{ and hence } a - b = O \\ &\iff O \in A \oplus -B \end{aligned}$$

□

In motion planning problems, the moving object B is usually constrained inside a workspace A. All the positions that B can access without going outside of A can be computed using the so-called "Minkowski decomposition" $A \ominus B$ [Ghosh, 1993], defined as:

$$A \ominus B = \bigcap_{b \in B} A_{-b}. \tag{2.1}$$

Note that in [Lozano-Pérez, 1983], $A \ominus B$ has a different meaning (defined as $A \oplus -B$). A 2D example of Minkowski decomposition is shown in Figure 2.2. The above definition (2.1) of Minkowski decomposition has a similar form to the definition of the Minkowski sum $A \oplus B = \bigcup_{b \in B} A_b$ in equation (1.6). The proposition below shows the relationship between the Minkowski decomposition and the Minkowski sum, and also explains why Minkowski decomposition can be used to find positions where B is completely contained in A.

Proposition 2.3. $A \ominus B = (A^c \oplus -B)^c$

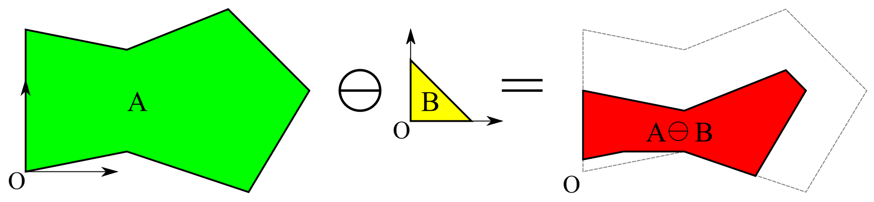


Figure 2.2: A 2D example of Minkowski decomposition.

Proof.

$$\begin{aligned} x \in (A^c \oplus -B)^c &\iff x \notin A^c \oplus -B \\ &\iff (B + x) \bigcap A^c = \emptyset \\ &\iff B + x \subseteq A \\ &\iff \forall b \in B, x + b \in A \\ &\iff \forall b \in B, x \in A_{-b} \\ &\iff x \in \bigcap_{b \in B} A_{-b} \end{aligned}$$

Thus since $\bigcap_{b\in B} A_{-b} = A \ominus B$, $A \ominus B = (A^c \oplus -B)^c$. □

2.2 Minkowski Sums of Convex Polyhedra

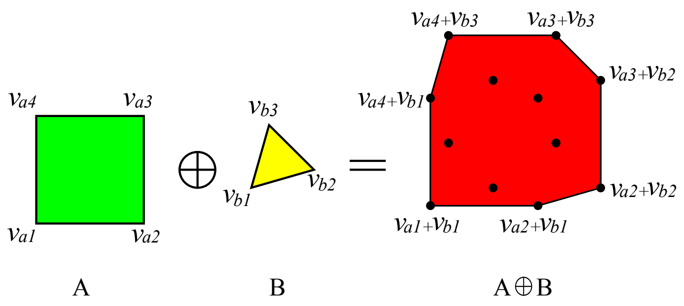


Figure 2.3: Convex hull approach for the Minkowski sum of two convex polygons. The black dots represent the vector sum of all the vertex pairs. The red polygon is their convex hull, and also the Minkowski sum.

Minkowski sums of convex polyhedra can be computed easily and efficiently. Convex hull or Gaussian map approaches are commonly used [Ghosh, 1993, Fogel and Halperin, 2007]. The convex hull approach first computes the vector sum between all possible pairs of vertices, one from each polyhedron, and then computes their convex hull (see Figure 2.3 for a 2D illustration). It is based on the following proposition, where $V_A = \{v_{a1}, v_{a2}, \cdots, v_{am}\}$ and $V_B = \{v_{b1}, v_{b2}, \cdots, v_{bn}\}$ denote the vertex sets of A and B respectively, and CH denotes the convex hull operator.

Proposition 2.4. *Suppose A and B are two convex polyhedra. Then $A \oplus B = CH(V_A \oplus V_B)$.*

Proof. We first prove $A \oplus B$ is convex. Suppose c_1 and c_2 are two arbitrary points in $A \oplus B$. We need to show for $0 \leq u \leq 1$, $uc_1 + (1-u)c_2$ is also in $A \oplus B$. Suppose $c_1 = a_1 + b_1$ and $c_2 = a_2 + b_2$, with a_1, $a_2 \in A$, and b_1, $b_2 \in B$. Then $uc_1 + (1-u)c_2 = u(a_1 + b_1) + (1-u)(a_2 + b_2) = (ua_1 + (1-u)a_2) + (ub_1 + (1-u)b_2)$. Since A and B are both convex, we have $ua_1 + (1-u)a_2 \in A$ and $ub_1 + (1-u)b_2 \in B$. Then $uc_1 + (1-u)c_2 \in A \oplus B$. This proves $A \oplus B$ is convex.

$V_A \oplus V_B$ is a set of discrete points. Since $V_A \subset A$ and $V_B \subset B$, we know $V_A \oplus V_B \subseteq A \oplus B$. Since $A \oplus B$ is convex and $CH(V_A \oplus V_B)$ is the smallest convex polyhedron containing $V_A \oplus V_B$, we have $CH(V_A \oplus V_B) \subseteq A \oplus B$.

On the other hand, if $A \oplus B \nsubseteq CH(V_A \oplus V_B)$, there must be a vertex of $A \oplus B$ that lies outside of $CH(V_A \oplus V_B)$ (otherwise, if all the vertices of $A \oplus B$ is inside $CH(V_A \oplus V_B)$, since both $A \oplus B$ and $CH(V_A \oplus V_B)$ are convex, we must have $A \oplus B \subseteq CH(V_A \oplus V_B)$). Denote this vertex as c and suppose $c = a + b$, $a \in A$ and $b \in B$. Since c is a vertex of $A \oplus B$, a and b must be vertices of A and B respectively, i.e., $a \in V_A$ and $b \in V_B$. (Note that since $A \oplus B$ is convex and c is one of its vertices, we can find a direction such that c is the unique maximum of $A \oplus B$ in this direction. Then a and b must also be the unique maxima of A and B respectively in this direction. This can be true only when a and b are vertices of A and B respectively.) Then $c = a + b \in V_A \oplus V_B$. This contradicts the assumption that c lies outside of $CH(V_A \oplus V_B)$. Then we must have $CH(V_A \oplus V_B) = A \oplus B$. □

In the convex hull approach, not all the points in $V_A \oplus V_B$ need be vertices of the Minkowski sum (see Figure 2.3); some of them are typically located in the interior and do not contribute to the Minkowski sum boundary. It is possible, by analyzing the outwardly oriented normal (simply called "outward normal" in the rest of this chapter) of each boundary facet, to directly find all the points in $V_A \oplus V_B$ that ultimately are the Minkowski sum vertices. This leads to the elegant Gaussian map approach, which is illustrated in Figure 2.4. A Gaussian map is a dual representation of a convex polygon or polyhedron. As seen in Figure 2.4, for a 2D polygon, a boundary edge maps to a point on the Gaussian circle (determined by its outward normal), and a vertex maps to an arc connecting the two points on the Gaussian circle corresponding to its two incident edges. For a 3D polyhedron, a boundary facet maps to a point on the Gaussian sphere (determined by its outward normal), a boundary edge maps to a geodesic arc connecting the two points corresponding to its two incident facets, and a vertex maps to a patch on the Gaussian sphere bounded by the geodesic arcs corresponding to its incident boundary edges. Overall, each point on the boundary of a convex polygon or polyhedron maps to a set of points (or a single point) on the Gaussian sphere that define the direction(s) in which the point on the boundary is maximal, and vice versa.

To compute the Minkowski sum of two convex polyhedra by the Gaussian map approach, we first construct their Gaussian spheres, and overlay these two Gaussian spheres to generate a new one. Then we can reconstruct the Minkowski sum from the new Gaussian sphere, as shown in Figure 2.4 for a 2D example. Many algorithms for computing Minkowski sums

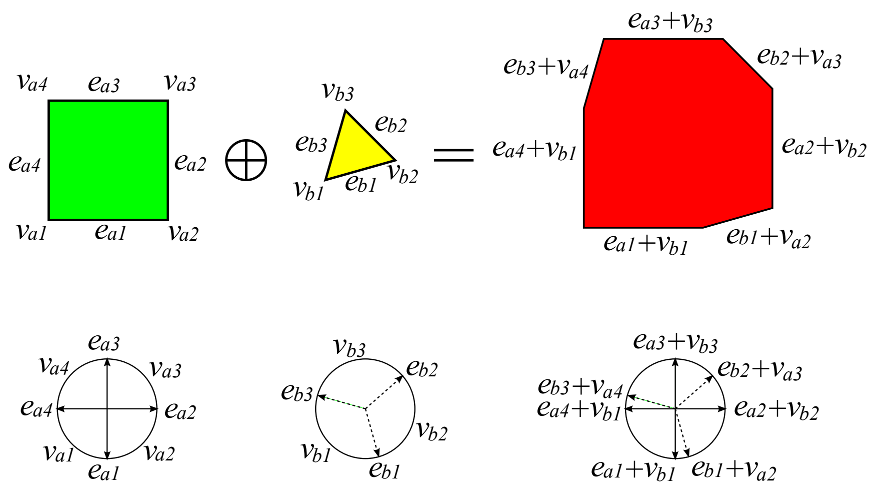


Figure 2.4: Gaussian map approach for the Minkowski sum of two convex polygons. The first row shows the Minkowski sum. The second row shows the two Gaussian circles and their overlay. A point on the overlay is generated by adding the two corresponding points on the two Gaussian circles.

of convex polyhedra are based on the Gaussian map idea. Fogel studied how to directly compute the arrangements of geodesic arcs embedded on the Gaussian sphere [Fogel, 2008]. In order to avoid computing arrangements on a sphere, Fogel and Halperin proposed using a “Cubical Gaussian Map,” where geodesic arcs on the Gaussian sphere are projected to the six faces of a bounding cube [Fogel and Halperin, 2007]. Barki et al. proposed the concept of “contributing vertices,” which is a variant of the Gaussian map approach [Barki et al., 2009b].

2.3 Minkowski Sums of Non-Convex Polyhedra

Although the Minkowski sum of two convex polyhedra has complexity of $O(mn)$ (here m and n denote the numbers of triangles of each input polyhedron) and can be computed easily and efficiently using either the convex hull or Gaussian map approach, the Minkowski sum of two non-convex polyhedra can have complexity as high as $O(m^3n^3)$ and becomes much more difficult to compute. In this section we review existing algorithms for non-convex objects. Most of them fall into two main categories: convex decomposition or convolution.

2.3.1 Convex-Decomposition Based Approaches

The basic idea of convex-decomposition based algorithms is decomposing the input non-convex polyhedra into convex pieces, computing all the pairwise Minkowski sums of these convex pieces, and then taking their union. It is based on the following property of Minkowski sums (distributivity):

Proposition 2.5. $(A \cup B) \oplus C = (A \oplus C) \cup (B \oplus C)$.

Proof.

$$\begin{array}{rl} & x \in (A \cup B) \oplus C \\ \Longleftrightarrow & x = y + c, y \in A \cup B, c \in C \\ \Longleftrightarrow & x = a + c \text{ or } b + c, a \in A, b \in B, c \in C \\ \Longleftrightarrow & x \in A \oplus C \text{ or } x \in B \oplus C \end{array}$$

Then $(A \cup B) \oplus C = (A \oplus C) \cup (B \oplus C)$. □

There are three steps in convex-decomposition based approaches: convex decomposition, computing the Minkowski sum of convex pieces, and union computation. While it is desirable to compute an optimal (minimum number of pieces) convex decomposition, this problem is known to be NP-hard [Chazelle, 1981]. Several practical algorithms are known to compute a sub-optimal or approximate convex decomposition [Chazelle et al., 1995, Ehmann and Lin, 2001, Lien and Amato, 2007]. The second step is typically performed using either the convex hull or Gaussian map approaches described in the previous section. The third step, computing the union of the intermediate Minkowski sum pieces, is the main bottleneck in implementing such convex-decomposition based algorithms, especially when the number of pairwise Minkowski sum pieces has quadratic complexity. Given n polyhedral objects, their union can have combinatorial complexity of $O(n^3)$ [Aronov et al., 1997].

Based on the algorithms used for convex decomposition and union computation, convex-decomposition based approaches for Minkowski sum computation can be either exact or approximate. The exact algorithms allow robust implementation and are able to find low dimensional boundaries, i.e., they are able to identify dangling faces or lines and singular points in the Minkowski sums [CGAL, 2008, Halperin, 2002, Hachenberger, 2009]. However, these algorithms are limited to relatively simple objects because of their performance. To compute the Minkowski sum of two polyhedra bounded by only hundreds of triangles, it usually takes tens of minutes [CGAL, 2011]. Varadhan and Manocha proposed another convex-decomposition based algorithm to compute an approximated boundary of Minkowski sums [Varadhan and Manocha, 2006]. Instead of computing the exact union of pairwise Minkowski sums, they compute a signed distance field and extract its zero iso-surface. Their algorithm provides geometrical and topological guarantees by using an adaptive subdivision algorithm. However, the performance of their algorithm is impacted by the large number of convex pieces after decomposition. The timing reported in their paper shows that computing the distance fields for tens of thousands of pairwise convex Minkowski sums usually takes quite a few minutes.

2.3.2 Convolution-Based Approaches

Convolution-based approaches have also been proposed for computing the boundary of Minkowski sums. Usually they start with a set of surface primitives that is a superset of the Minkowski sum boundary. These surface primitives are then trimmed and filtered to form the final boundary.

For objects with smooth boundary surfaces, their convolution is well defined. If we denote the convolution of two surfaces as $\star$, then the convolution of two smooth boundary surfaces ∂A and ∂B is defined as

$$\partial A \star \partial B = \{a + b \mid a \in \partial A, b \in \partial B, n_a = n_b\}, \tag{2.2}$$

where n_a denotes the unit outward normal at point a. A 2D example of the convolution of two smooth curves is shown in Figure 2.5. Note that the convolution in this context is different from its usual definition of "convolving two functions." For two objects the convolution of their boundaries is a superset of the boundary of their Minkowski sum and also a subset of the Minkowski sum of their boundaries [Mühlthaler and Pottmann, 2003, Peternell and Steiner, 2007]; i.e.,

$$\partial(A \oplus B) \subseteq \partial A \star \partial B \subseteq \partial A \oplus \partial B. \tag{2.3}$$

If both objects are convex, it is easy to show that $\partial A \star \partial B = \partial(A \oplus B)$ [Ghosh, 1993]. If at least one of them is non-convex, the convolution may become self-intersecting (see Figure 2.5 for an example), and the Minkowski sum boundary can be extracted by trimming and filtering the convolution surfaces.

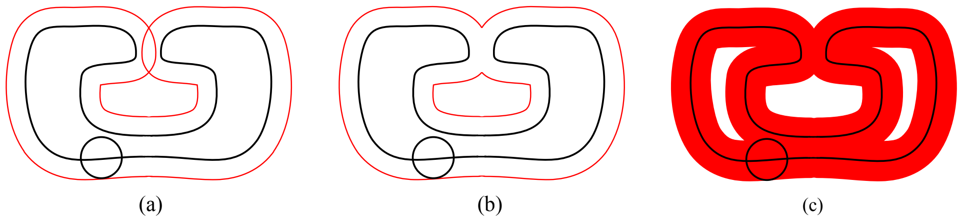


Figure 2.5: Convolution of two smooth curves. (a) Two smooth curves (in black) and their convolution (in red). (b) The Minkowski sum boundary of the two shapes (in red). (c) The Minkowski sum of the two curves (in red).

Polyhedra do not have smooth boundary surfaces, and the outward normals at their vertices and edges are not well defined. The definition of convolution for polyhedra is much more complicated and sometimes confusing, especially for non-convex polyhedra. In [Ghosh, 1993], the convolution of two polyhedra is defined using the concept of Gaussian map (called "slope diagrams" in [Ghosh, 1993]), and for non-convex input, an additional concept called "negative objects." In that definition, the convolution of two polyhedra can be seen as an extension of equation (2.2), which defines the convolution of smooth boundary surfaces, where n_a and n_b are replaced with the corresponding features (a single point, a geodesic arc, or a patch, depending on the position of a or b on the boundary) to which points a and b are mapped to on the Gaussian sphere, and then instead of checking whether the two outward normals are the same, the intersection of the two features is checked.

Computing the exact convolution of two polyhedra is difficult itself since it involves 3D arrangements and Boolean operations. Usually people compute the exact convolution only if both input models have smooth boundaries and if explicit parameterization of both boundaries exists [Mühlthaler and Pottmann, 2003, Peternell and Steiner, 2007]. Many convolution-based approaches for polyhedra just start with $\partial A \oplus \partial B$ as the convolution [Lien, 2008d, Kaul and Rossignac, 1992], and then filter and trim its surface primitives to extract the final Minkowski sum boundary. The filtering and trimming operations may become very complex since the number of generated surface primitives has quadratic complexity and they may intersect each other arbitrarily in 3D space. The algorithm we shall describe in Chapter 5 uses this basic approach, but we use GPUs to directly compute a voxelization so we can avoid the complex 3D computations.

Below we briefly discuss some existing convolution-based algorithms. Guibas and Seidel presented an output sensitive algorithm for computing the convolution of 2D curves [Guibas and Seidel, 1986]. Kaul and Rossignac introduced a set of criteria to cull out facets that are not part of the Minkowski sum boundary [Kaul and Rossignac, 1992]. These criteria were used later in other works [Lien, 2008a,d, Liu et al., 2009], and also in this dissertation (Section 5.2.1). Peternell and Steiner studied how to extract the Minkowski sum boundary from the convolution of two objects with piecewise smooth boundaries [Peternell and Steiner, 2007]. As discussed in the previous paragraph, Lien started with a brute force convolution, which is simply defined as $\partial A \oplus \partial B$, and then computed facet-facet intersections as 2D arrangements on each facet [Lien, 2008d]. Then he introduced the novel idea of using collision detection tests to merge and filter cells from 2D arrangements. Unfortunately the 2D arrangements and collision detection become both time and memory consuming when the size and complexity of the input models increase.

2.3.3 Other Approaches

To overcome the computational complexity introduced by 3D operations, some approaches seek to use other lower dimensional representations. Voelcker and his group suggested using "ray representations" (ray-reps) to reduce 3D Minkowski sum computation to 1D Boolean operations [Menon and Voelcker, 1993, Hartquist et al., 1999], but no practical implementation of this algorithm is known. Lien proposed a point-based approach that creates a point set covering the Minkowski sum boundary [Lien, 2008a]. This approach starts with a set of points both on and inside the Minkowski sum boundary. Several filters, including the ones introduced in [Kaul and Rossignac, 1992], are used to cull out points that are not on the boundary to generate the final point set. The main drawback of this approach is that its result includes no information of the interior of the Minkowski sum. Kavraki proposed

#tri 1	#tri 2	Timing (min:sec)
448	6	3:24
448	128	9:13
336	24	14:42
1,000	1,000	164:11

Table 2.1: Performance of 3D Minkowski sum computation for CGAL library. The Minkowski sums were computed with CGAL 3.3 on a machine with a 2.4 GHz AMD Opteron processor and 4 GB RAM [CGAL, 2011].

voxelizing both input models, then converting the Minkowski sum to the convolution (the usual mathematical convolution, not the convolution of two surfaces mentioned in the previous section) of two 3D arrays representing the voxels, and finally computing the convolution using a fast Fourier transform (FFT) [Kavraki, 1995, Lysenko et al., 2010, 2011]. The main drawback of this approach is the low resolution (128^3 or 256^3) used for the voxelization due to memory limitations, since each voxel needs to be represented as a floating point number.

Some algorithms have also been introduced for handling specific types of objects. Seong et al. presented an algorithm for computing Minkowski sums of surfaces generated by slope-monotone closed curves [Seong et al., 2002]. Mühlthaler and Pottmann introduced an explicit parameterization of the convolution of two ruled surfaces [Mühlthaler and Pottmann, 2003], which was defined in equation (2.2). Barki et al. proposed an approach for computing the Minkowski sum of a convex polyhedron and a non-convex polyhedron whose boundary is completely recoverable from three orthogonal projections [Barki et al., 2009a]. Other authors have introduced fast algorithms for the special case of rotating convex polyhedra by tracking the changes to the topology of the arrangement on the Gaussian sphere [Lien, 2008b, Behar and Lien, 2011, Mayer et al., 2010].

2.3.4 Performance of Existing Libraries

There are two publicly available libraries for computing general 3D Minkowski sums, the CGAL library [CGAL, 2008] and "m+3d" provided by Lien [Lien, 2008d]. The former is based on convex decomposition and the latter based on convolution. Table 2.1 and 2.2 give the reported timings for both libraries. Note that the performance is determined not only by the numbers of triangles of input models, but also by their specific shapes. Overall m+3d has better performance compared to CGAL, but CGAL uses exact arithmetic so that it is able to find low dimensional boundaries, i.e., to identify dangling faces or lines and singular points in the Minkowski sums. From the two tables, we can see that for input models with hundreds or thousands of triangles, both libraries take minutes or even tens of minutes to compute their Minkowski sums.

#tri 1	#tri 2	Timing (min:sec)
540	942	5:19
2,116	992	5:47
12,396	992	12:35
32,236	992	15:21

Table 2.2: Performance of 3D Minkowski sum computation for m+3d. The timings were obtained on a PC with two Intel Core 2 CPUs at 2.13 GHz with 4 GB RAM [Lien, 2008d].

2.4 GPU-Based Voxelization

In chapter 4 and 5, we will present two GPU-based algorithms for directly voxelizing the Minkowski sum of two watertight polyhedra. In this section we briefly review several GPU-based voxelization algorithms that are related to the techniques we will use in these chapters.

First we give an overall introduction to voxelized representations and voxelization algorithms. Voxelized representations (see Figure 1.10 for an example) are popular because of their simplicity and regularity [Gibson, 1995]. A variety of fields exploit voxelized representations including virtual medicine [Kreeger and Kaufman, 1999], haptic rendering [McNeely et al., 1999], shadow rendering [Kim and Neumann, 2001], CSG operations [Fang and Liao, 2000], and visibility queries [Schaufler et al., 2000]. Voxelization is the process of generating a voxelized representation for geometric objects. Voxelization algorithms can be classified into *surface voxelization* or *solid voxelization*, depending on whether they voxelize only the boundary surface or the whole interior. Another classification is *binary voxelization*, where each voxel is represented by 0 or 1, or *non-binary voxelization*, where each voxel is represented by a real value in the range $[0, 1]$. In this dissertation, we only consider binary voxelizations.

Most voxelization algorithms are GPU-based due to the GPU's built-in rasterization capability. Karabassi et al. presented a depth buffer based voxelization algorithm [Karabassi et al., 1999], which works only for an object whose boundary can be completely seen from the six orthogonal directions. The input object is projected to the six faces of its bounding box and depth information is then read back from the depth buffer and used to reconstruct the object. Llamas presented an algorithm for solid voxelization inspired by the stencil shadow volume technique [Llamas, 2007]. In this technique, the stencil buffer is increased by one for back faces and decreased by one for front faces (both with wrapping enabled to avoid saturation); thus after rendering a slice of the input object, any voxel centered at this slice and located inside the object will have a non-zero stencil value. This algorithm can also be used to voxelize the union of a set of watertight objects. It will be discussed in more details in section 4.2 and 4.3. In the algorithm proposed by Fang and Chen, the object is rendered slice by slice along the z direction, and each slice is voxelized individually [Fang and Chen, 2000]. This algorithm directly creates a surface voxelization, from which a solid voxelization can then be generated using a parity check. However, surfaces parallel or nearly parallel to the projection direction are not completely voxelized, and the memory cost for a high resolution volume is high since each voxel requires a byte (rather than a bit) of memory. To address these issues, Dong et al. proposed projecting the model along three orthogonal directions and encoding multiple voxels in one texel (texture element) [Dong et al., 2004]. We will use the same approach in the algorithm described in section 5.3.1.

Chapter 3

2D Accessibility Analysis for Waterjet Cleaning

In section 1.3.2, we summarized the relationship between Minkowski sums and the widely used configuration space (C-space) approach for motion planning. In this chapter, we introduce a C-space approach to analyzing the accessibility and cleanability problem for waterjet cleaning.

Effective cleaning with high pressure waterjets requires direct impact of jets and sufficiently high impact pressure. The objective of this research is to find all such cleanable regions, given a CAD model of a workpiece, by means of geometric accessibility analysis. We use a C-space approach for addressing the problems of both optimum surface proximity for effective cleaning and collision avoidance between the cleaning lance and the workpiece. Minkowski sums are used to compute the C-spaces and cleanable regions are then found by visibility analysis. Implementations and results for 2D examples are shown to validate the approach.

3.1 Introduction and Problem Statement

Cleaning engine components to remove hard particle contaminants introduced during the manufacturing process is becoming a significant issue for industry [Ávila et al., 2006, Berger, 2006]. Examples of mechanical components that are particularly prone to solid particle contamination are engine blocks and cylinder heads fabricated by sand casting wherein the sand particles from the mold get embedded onto the surface of the cast product. A commonly used cleaning method is High Pressure Jets (HPJs), where the cleaning action is obtained from the impact of a high speed jet of water. Experience in the automotive industry has shown that efficacy of HPJ cleaning is strongly dependent on the geometry of the workpiece [Arbelaez et al., 2008].

Using HPJ cleaning for the removal of embedded particles such as casting sand, it has been found that direct impact of the jets is often necessary for their removal [Ávila et al., 2006]. Hence, the HPJ lance — a rigid tube with a cleaning nozzle on the end — must be able to access the features such that the entire surface area receives a direct impact from the jets. Another necessary condition is sufficiently high impact pressure on the workpiece surface when it is hit by the waterjets from the nozzle [Ávila, 2007]. If the distance between the nozzle and the surface becomes large enough, the impact pressure drops below the minimum requirement for effective cleaning. In this research we take into consideration both the ability to achieve direct impact and the magnitude of impact pressure. We say a region of the surface is *cleanable* if waterjets can directly access it and the impact pressure is high enough to perform effective cleaning. Otherwise we say it is *non-cleanable*. Figure 3.1 illustrates these two types of non-cleanable regions.

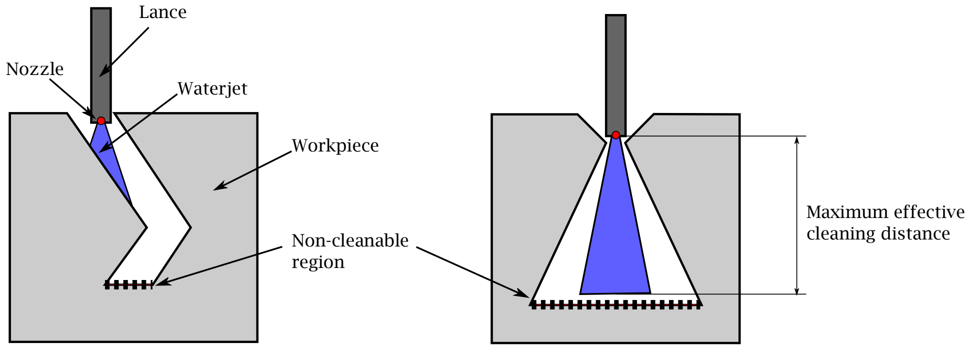


Figure 3.1: Two types of non-cleanable regions (dotted). On the left the surface is not directly accessible by the waterjet. On the right the distance between the nozzle and the surface is too large to provide enough impact pressure for effective cleaning.

In this research we present an approach for finding all possible cleanable regions of a workpiece by means of geometric accessibility analysis. We consider a simplified 2D problem where the workpiece is represented as a polygon. We also assume that the workpiece position and orientation is fixed and that the lance can only translate (not rotate) relative to the workpiece during the cleaning process. We only consider the collision between the lance and workpiece and do not consider the lance holder and other obstacles in our implementation, although our algorithm could be trivially extended to incorporate these inputs as well. An example result of our algorithm is shown in Figure 3.2. Although the algorithm is demonstrated in 2D, it can be extended to 3D as well.

Our algorithm consists of two major stages. In the first stage, the entire C-space is divided into three disjoint C-spaces — cleaning C-space, non-cleaning C-space, and interference C-space — by using Minkowski sums. Then in the second stage, we find all the regions of the workpiece that are cleanable when the lance is positioned in the cleaning C-space, based on visibility and effective cleaning distance.

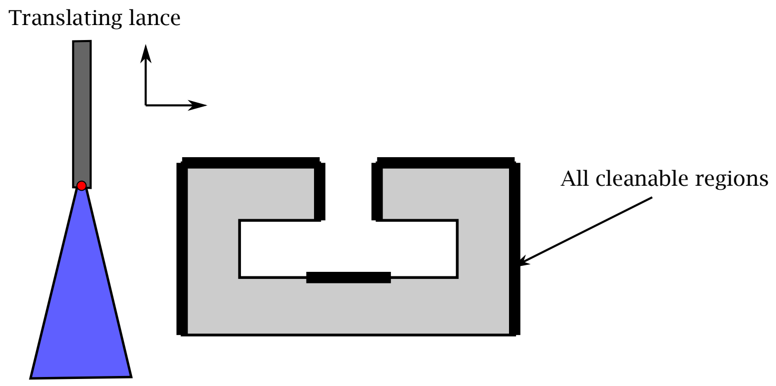


Figure 3.2: An example of all possible cleanable regions (thickened). The workpiece is fixed and the lance can only translate.

3.2 Related Work

C-spaces have been heavily used for collision avoidance in automatic path planning. Lozano-Pérez first introduced the C-space approach for motion planning of a rigid object among physical obstacles [Lozano-Pérez, 1983]. Every point in the C-space corresponds to a set of independent parameters that characterize the position and orientation of the rigid object. Choi et al. applied the C-space approach to 3-axis NC tool path generation for sculptured surface machining [Choi et al., 1997]. Tool paths generated from the C-spaces were gouge-free and collision-free.

Accessibility analysis has been recognized as an important tool in various applications such as dimensional inspection, machining, assembly, and mold design. In machining, for example, accessibility analysis helps in determining machinability by finding the set of directions from which the part may be approached by the cutting tool. Woo and his group introduced the concept of spherical visibility maps and applied them to various manufacturing problems like optimal workpiece orientation [Chen et al., 1993, Woo, 1994]. Spitz et al. presented a discrete approximation algorithm for computing accessibility information by exploiting computer graphics hardware [Spitz et al., 1998].

3.3 Mathematical Model of Waterjet Cleaning

Before we analyze the accessibility of waterjets, we need to compute the *effective water zone* — the water zone where the impact pressure is high enough for effectively cleaning. The impact pressure becomes low where the distance to the nozzle is large and where waterjets mix with the surrounding air. To compute the impact pressure inside the waterjets, we use a mathematical model introduced in [Leu et al., 1998]. Let the nozzle center be the origin O. The impact pressure P at some point inside the waterjets is a function of the standoff distance x and radial distance r, as depicted in Figure 3.3(a).

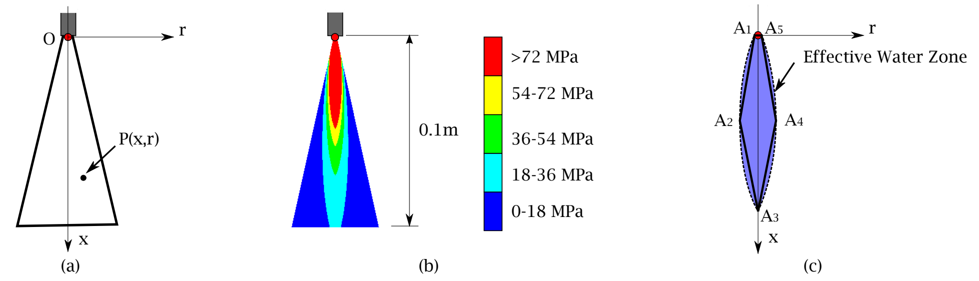


Figure 3.3: Impact pressure of a waterjet. (a) Impact pressure P is a function of standoff distance x and radial distance r. (b) An example of impact pressure distribution inside the waterjets. (c) The effective water zone is approximated as a polygon $A_1A_2A_3A_4A_5$.

According to [Leu et al., 1998], the impact pressure can be expressed as:

$$P(x,r) = \left(\frac{D}{x}\right)^2 \left[1 - \left(\frac{r}{Cx}\right)^{1.5}\right]^3 \tag{3.1}$$

where C is the spreading coefficient of water, and D is a parameter determined by the water pressure from the pump, the nozzle radius, and some physical constants of water. Both C and D can be assumed as constants during the cleaning process. Based on this equation, we can compute the impact pressure distribution inside the waterjets. Figure 3.3(b) illustrates such an example where the water pressure from the pump is 200MPa and the nozzle radius is 0.3mm. If the minimum effective impact pressure is P_{min}, then the effective water zone is the set of all locations satisfying $P \geq P_{min}$. We can see from Figure 3.3(b) that iso-pressure curves have an ellipse-like shape. For simplicity we approximate the effective water zone as a polygon $A_1A_2A_3A_4A_5$, as shown in Figure 3.3(c). The short edge A_1A_5 represents the width of the nozzle. Since usually the nozzle radius r_0 is very small and the impact pressure near the nozzle is very high, we assume the effective water zone emanates from the full width of the nozzle and set the coordinates of A_1 and A_5 as:

$$\begin{aligned}(r_1, x_1) &= (-r_0, 0),\\ (r_5, x_5) &= (r_0, 0).\end{aligned} \tag{3.2}$$

The coordinates of A_2 and A_4 are determined by the system of equations:

$$\begin{aligned}&\partial P/\partial x = 0,\\ &P = P_{min};\end{aligned} \tag{3.3}$$

and the coordinates of A_3 are determined by the system of equations:

$$\begin{aligned}&r = 0,\\ &P = P_{min}.\end{aligned} \tag{3.4}$$

Solving these equations we obtain the coordinates of A_2, A_3, and A_4 as:

$$\begin{aligned}
(r_2, x_2) &= \left(-0.26\frac{CD}{\sqrt{P_{min}}}, 0.58\frac{D}{\sqrt{P_{min}}}\right), \\
(r_3, x_3) &= \left(0, \frac{D}{\sqrt{P_{min}}}\right), \\
(r_4, x_4) &= \left(0.26\frac{CD}{\sqrt{P_{min}}}, 0.58\frac{D}{\sqrt{P_{min}}}\right).
\end{aligned} \tag{3.5}$$

In our algorithm implementation, we use the polygon $A_1A_2A_3A_4A_5$ as the effective water zone. Slightly more accurate simulations could be performed using a convex n-gon with more sides, at the expense of longer running times.

3.4 C-space Computation

As the first stage of our algorithm, we divide the entire configuration space into three subsets based on the geometry of the lance, the workpiece, and the effective water zone. A configuration of an object is defined by independent parameter values that characterize the position the object [Lozano-Pérez, 1983]. In our simplified cleaning problem, the lance system is a rigid polygon that can only translate in the plane, so each configuration has two degrees of freedom. We pick a point in the polygon as the *reference point*, and use its x and y coordinates to specify the corresponding configuration. The set of all configurations is the configuration space (C-space). Some configurations are forbidden during the cleaning process because the lance will collide with the workpiece under these configurations. The others are safe; however, some of them are not valid for cleaning if the effective water zone doesn't "touch" some region of the surface of the workpiece. (Note that not all surfaces touched by the efficient water zone will necessarily be cleaned in the corresponding configuration due to visibility constraints, as described in the next section.) Thus we can divide the whole C-space of waterjet cleaning into three disjoint C-spaces:

- *Interference C-space* (A_I): the C-space where the lance collides with the workpiece;
- *Non-cleaning C-space* (A_N): the C-space where the lance does not collide with the workpiece but the waterjets fail to effectively clean the workpiece;
- *Cleaning C-space* (A_C): the C-space where the waterjets can effectively clean some regions of the workpiece.

We represent the lance, the effective water zone, and the workpiece as polygons L, W, and P respectively, and place the reference point O at the center of the water nozzle, as shown in Figure 3.4 (a). We also assume that the workpiece does not have any self-intersections or enclosed holes (enclosed holes, if any, can be marked as non-cleanable and removed since the lance can never go into these holes). Since we don't consider the lance holder geometry in our problem, we model the lance as a rectangle with infinite height to simulate the existence of the

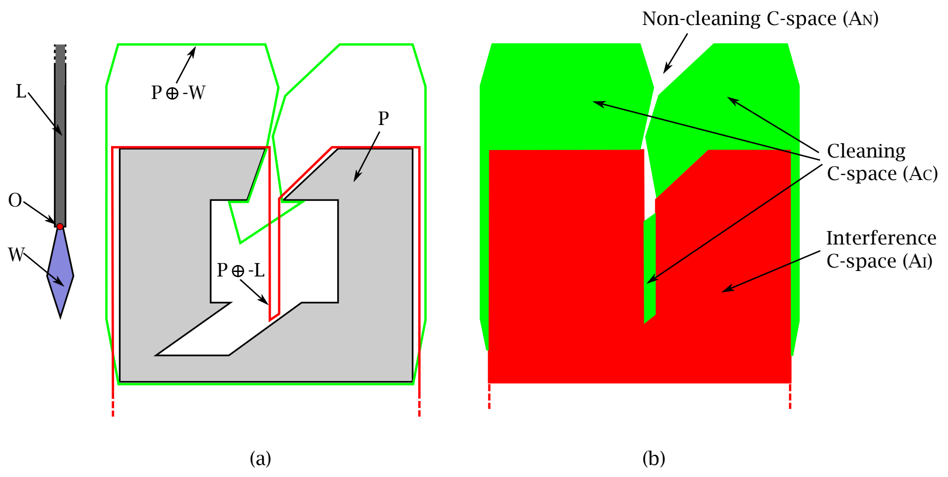


Figure 3.4: Computation of the cleaning, non-cleaning, and interference C-spaces. O is the reference point.

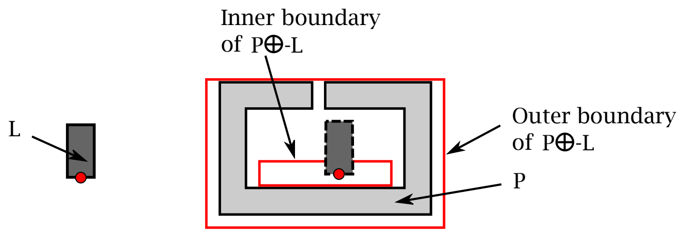


Figure 3.5: A hole in the interference C-space. The lance cannot access holes in $P \oplus -L$, such as in the dashed configuration pictured, although these configurations would not be in the interference C-space.

lance holder. In practice a height larger than the diagonal of the workpiece's bounding box is sufficient. (Alternatively, a polygon with more sides could be used to accurately represent the combined lance and lance holder geometry.) Computation of the three C-spaces can be described using Minkowski sums as follows:

$$\begin{aligned} A_I &= P \oplus -L \\ A_C &= (P \oplus -W) \setminus A_I \\ A_N &= (A_I \cup A_C)^c \,. \end{aligned} \tag{3.6}$$

In the above equation, the interference C-space A_I is the Minkowski sum of the workpiece P and the inverse lance $-L$. Since L effectively has an infinite height, there will be no holes in $P \oplus -L$ (because a ray emanating from any point in $P \oplus -L$ and going down vertically also lies in $P \oplus -L$). Thus we avoid the case where the lance can be "placed" in cavities without collision but cannot access them from the outside (see Figure 3.5). The cleaning C-space A_C

is the difference between $P \oplus -W$ and A_I. The non-cleaning C-space A_N is the complement of the union of the cleaning and the interference C-space. The result of computing these three C-spaces is illustrated in Figure 3.4 (b), where the (light) green area is the cleaning C-space and the (dark) red area is the interference C-space.

3.5 Algorithms for Finding Cleanable Regions

As the second stage of our algorithm, we find all the cleanable regions, building on the computation of the three C-spaces, which give us information about where we can place the lance for effective cleaning and where the lance collides with the workpiece. Under any configuration in the cleaning C-space, the waterjets clean some regions of the workpiece. In this section, we describe an algorithm for finding all such cleanable regions of a workpiece, given a translating lance. We solve this problem in two steps. First we find the cleanable regions under a specific configuration (lance position), taking visibility into account (section 3.5.1), and then we use a simulation approach to find all such cleanable regions (section 3.5.2).

3.5.1 Finding Cleanable Regions under a Specific Configuration

For a configuration $c \in A_C$, we want to find the cleanable regions when the lance is positioned in this configuration. These cleanable regions are a subset of the surface regions that lie inside the effective water zone, since they must also be visible from the nozzle. The actual nozzle has a radius that is usually very small compared to the size of the lance or the workpiece. In our algorithm, we test the visibility from point O', the intersection of lines A_2A_1 and A_4A_5 bounding the effective water zone (see Figure 3.6 (a)).

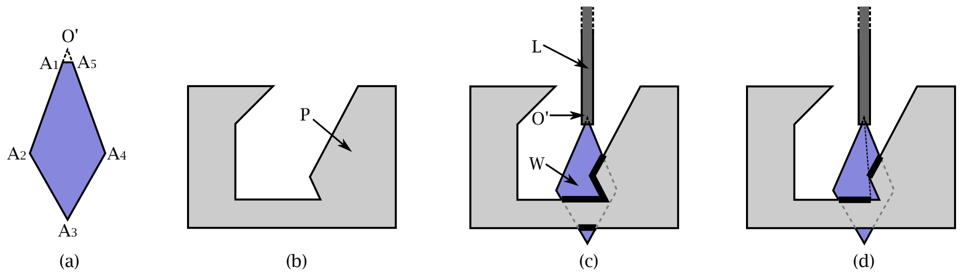


Figure 3.6: Computation of the cleanable regions under a specific configuration. (a) The effective water zone. (b) The workpiece to be cleaned. (c) Surfaces inside the effective water zone. (d) The subset of such surfaces also visible from the nozzle.

The algorithm for finding cleanable regions under a specific configuration consists of three steps:

1. Compute the regions of the workpiece that lie inside the effective water zone, which is a set of line segments, denoted as R_E (highlighted in Figure 3.6 (c));
2. For each line segment in the set R_E, compute the portion of it that is visible from the nozzle (highlighted in Figure 3.6 (d));
3. The cleanable regions under this configuration are the union of all such visible sub-segments.

For the first step, we can use either a naïve approach to compute the intersections between each edge of the workpiece and the effective water zone, or a more efficient one with hierarchical data structures such as quadtrees. Currently we simply use a naïve approach. For the second step, we first give a clear definition of visibility to handle ambiguous cases. We say a point Q is *visible* from point O' if the line segment $O'Q$ doesn't intersect the interior of the workpiece. For simplicity, in the remainder of this chapter, when we say a point or a region is visible, we mean that it is visible from the point O'. According to this definition, points A and D in Figure 3.7 (a) are not visible, but points B and C are visible. We give the algorithm for computing the visibility of a point below.

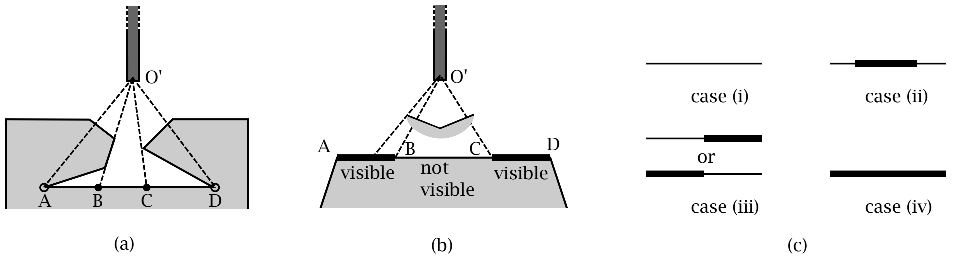


Figure 3.7: Visible points and visible regions. (a) Points A and D are not visible; point B and C are visible. (b) Disjoint visible regions on a line segment. (c) Four different visibility cases for a line segment.

The visible regions on a line segment are the set of all visible points on it. We can prove by contradiction that for a line segment on the workpiece, its visible regions are connected.

Proposition 3.1. *The visible regions of a line segment on the workpiece are connected.*

Proof. Suppose that on a line segment AD, there exist disjoint visible regions AB and CD (Figure 3.7 (b)). Since BC is not visible, there must exist a region of the workpiece inside $\triangle O'BC$ that blocks the visibility. Since the workpiece is a polygon without holes, the boundary of this region should be connected with line segment AD, and it cannot cross line segment BC (otherwise the workpiece is self-intersecting). It cannot go through point O' either, otherwise the lance would be colliding with the workpiece. So this region of the workpiece inside $\triangle O'BC$ has to cross the interior of line segment $O'B$ or $O'C$, which causes

Algorithm 3.1 Compute the visibility of point Q

```
1: Find all intersection points between the line segment $O'Q$ and the boundary of the
   workpiece. (If an edge of the workpiece overlaps $O'Q$, only its endpoints are deemed
   potential intersection points.)
2: Store these intersection points in a list $L$ and sort them lexicographically by coordinates.
3: visibility ← true
4: for each pair of adjacent points $L[i]$, $L[i+1]$ in $L$ do
5:    $M \leftarrow (L[i] + L[i+1])/2$
6:    if $M$ is in the interior of the workpiece then
7:       visibility ← false
8:    end if
9: end for
10: return visibility
```

the neighborhoods of B or C to be invisible. This contradicts the assumption that the whole of line segments AB and CD are visible. □

Based on this theorem, we can conclude that any line segment on the workpiece belongs to exactly one of the four different visibility cases shown in Figure 3.7 (c): the whole line segment is invisible (case *i*), a region in the interior is visible (case *ii*), a region on one end is visible (case *iii*), or the whole line segment is visible (case *iv*). Thus, to compute the visible regions of a line segment, we first find the visible vertices of the workpiece that occlude the line segment from the nozzle (i.e., inside the triangle formed by the line segment and O' on the nozzle), if any. We then compute the distinct intersection(s) between the ray(s) emanating from the nozzle that pass(es) through these vertices, and the line segment. If the number of distinct visible intersection points is one or two, the line segment belongs to case (*iii*) or case (*ii*), respectively. If zero, it is either case (*i*) or case (*iv*), which can in turn be differentiated by the visibility of the middle point of the line segment. The detailed algorithm is described below.

We apply this algorithm to each line segment in the set R_E to find its visible regions, and then take the union of all these visible regions. The union is the cleanable regions under the *current* configuration.

3.5.2 Finding All Cleanable Regions

All possible cleanable regions are the union of the cleanable regions when the lance is positioned under each configuration of the cleaning C-space. A naïve approach might sample the whole cleaning C-space, calculate the cleanable regions under each sample configuration, and then take their union. However, we can prove that it is sufficient to sample only the boundary between the cleaning and the interference C-spaces to cover all the possible

Algorithm 3.2 Compute the visible regions of a line segment Q_1Q_2 on the workpiece

```
1: Find all visible vertices of the workpiece, and store them in an array $A_{vert}$.
2: Initialize an empty point array $A_{pnt}$.
3: for all $V \in A_{vert}$ do
4:    if ray $O'V$ intersects the line segment $Q_1Q_2$ at point $T$ and $T$ is visible then
5:       add $T$ to $A_{pnt}$
6:    end if
7: end for
8: Remove duplicate points in $A_{pnt}$.
9: if $size(A_{pnt})=1$ then {this is case (iii)}
10:    if $Q_1$ is visible then
11:       return line segment ($Q_1$, $A_{pnt}[0]$)
12:    else
13:       return line segment ($A_{pnt}[0]$, $Q_2$)
14:    end if
15: else if $size(A_{pnt})=2$ then {this is case (ii)}
16:    return line segment ($A_{pnt}[0]$, $A_{pnt}[1]$)
17: else if $size(A_{pnt})=0$ then
18:    $M \leftarrow (Q_1 + Q_2)/2$
19:    if $M$ is visible then {this is case (iv)}
20:       return line segment ($Q_1$, $Q_2$)
21:    else {this is case (i)}
22:       return NULL
23:    end if
24: else
25:    assert(false) {this should not happen}
26: end if
```

cleanable regions.

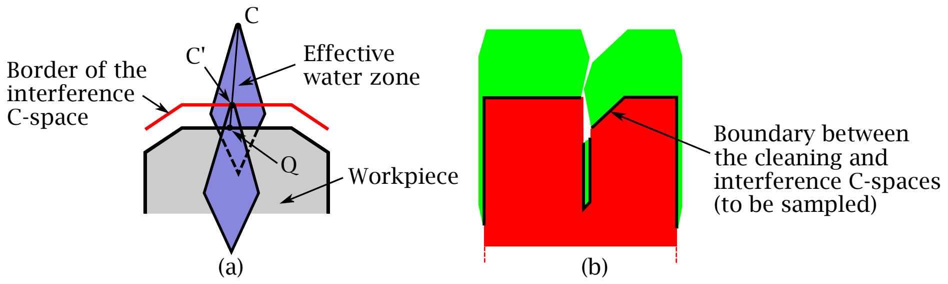

C
C'
Border of the interference C-space
Effective water zone
Q
Workpiece
(a)
Boundary between the cleaning and interference C-spaces (to be sampled)
(b)

Figure 3.8: Only the boundary between the cleaning and interference C-spaces needs to be sampled.

Proposition 3.2. *If a point on the workpiece is cleanable, then it is cleanable under some configuration on the boundary between the cleaning and interference C-spaces.*

Proof. Suppose that a point Q on the workpiece is cleanable. There must exist a configuration C in the cleaning C-space such that Q can be cleaned when the lance is positioned at C (see Figure 3.8 (a)). Since Q lies inside the interference C-space and C lies outside, the line segment QC must intersect the border of the interference C-space. Call the intersection point C' (if more than one intersection point exists, we pick the one closest to C). Because of the convexity of the effective water zone, Q still lies in the effective water zone if the lance is moved from C to C'. Since Q is visible from C, it is also visible from C'. So Q can be cleaned under configuration C'. Since C' lies on the border of the interference C-space and the lance positioned at it cleans part of the workpiece, C' must belong to the boundary between the interference and cleaning C-spaces. □

For the example in Figure 3.4, the boundary between its cleaning and interference C-spaces are shown in Figure 3.8 (b). To compute the boundary, we first compute the border of $P \oplus -L$ (see Figure 3.4 (a)), then find the portions that lie inside $P \oplus -W$. In practice, we offset the boundary into the cleaning C-space for a very small distance to avoid the situation where a portion of the boundary may overlap the edges of the workpiece. We then regularly sample points along the boundary, compute the visible regions for each sample point using the algorithm described in section 3.5.1, and take the union of all these visible regions.

3.6 Implementation

We implemented all the above algorithms using the CGAL library [CGAL, 2008], which offers robust implementations of some basic algorithms like Minkowski sums, point membership classification (PMC), intersection point computation, etc. We used the predefined Cartesian kernel with exact predicates and constructions to guarantee robustness when handling corner cases like the visibility computation for points B and C in Figure 3.7 (a) [Halperin, 2002].

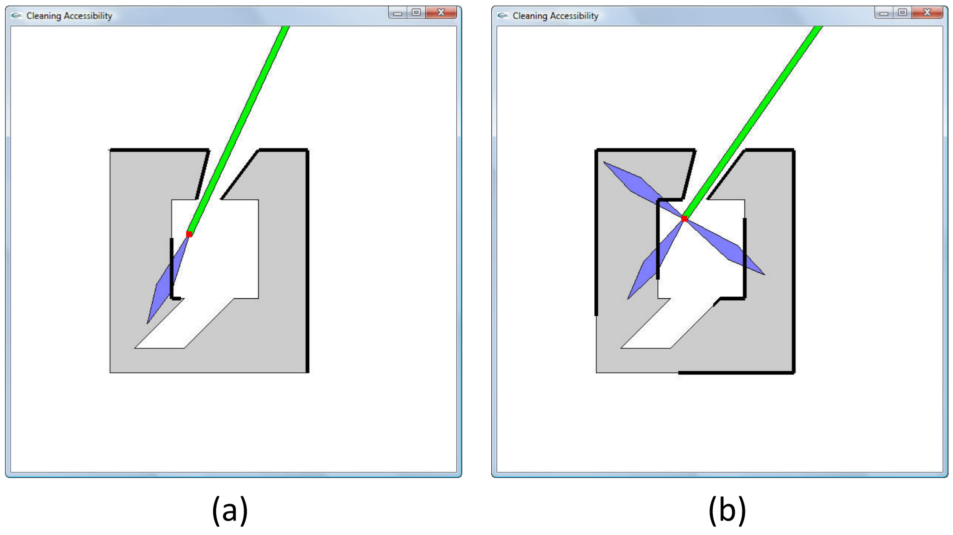


Figure 3.9: Example results of our program. All the possible cleanable regions for a slanted (a) single-orifice and (b) multi-orifice lance are highlighted.

In our interactive program, the user can translate the lance or change its orientation using the keyboard. The color of the lance indicates the current C-space it lies in: red means the interference C-space, green means the cleaning C-space, and gray means the non-cleaning C-space. The cleanable regions under the current configuration are highlighted. The user can also invoke the simulation to find all possible cleanable regions. Our program supports multi-orifice nozzles as well. Two examples of our results are shown in Figure 3.9. The accuracy of the results is determined by the sampling frequency on the boundary between interference and cleaning C-spaces. In our implementation, a fixed sampling step equal to the nozzle radius r_0 (see Equation (3.2)) was used to prevent unintended gaps between the cleanable regions that could otherwise cause artifacts due to the discrete sampling.

3.7 Discussion

An exact algorithm for finding all the cleanable regions remains as future work. A promising approach would find the exact locations of “critical points” on the boundary of the workpiece (the endpoints of the highlighted line segments in Figure 3.9). To find these critical points directly would be difficult; instead we find a set of possible candidate points that is a superset of all the actual critical points. Such candidates subdivide the boundary of the workpiece into many line segments. Inside each line segment, all points have the same cleanability, i.e., they are all either cleanable or non-cleanable. To find the possible candidates for critical

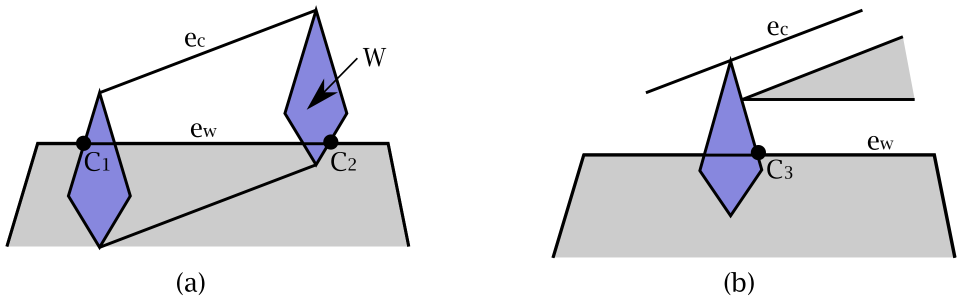


Figure 3.10: Possible candidates for critical points.

points, we consider each pair consisting of an edge (e_w) of the workpiece and an edge (e_c) of the boundary between the interference and cleaning C-space. In addition to the original vertices of the workpiece, possible candidates include, but are not limited to:

- Intersection points between e_w and the boundary of $e_c \oplus W$, such as C_1 and C_2 in Figure 3.10 (a);
- Extreme points caused by the occlusion of another vertex on the workpiece, such as C_3 in Figure 3.10 (b).

Conditions for finding the complete set of candidates remain future work.

More accurate geometries can be used for the effective water zone W and the lance L. A more accurate approximation of the effective water zone (Figure 3.3) can be used as long as it is a convex polygon (required by Proposition 3.2). Currently the geometry of the lance holder is not considered and the lance is modeled as a simple rectangle. We can use a more complex polygon to accurately represent the combined geometry of the lance and lance holder; this polygon needs not be convex.

The approach described in this paper can be extended to 3D as well. Algorithm 3.1 for computing the visibility of a point still holds in 3D, but Proposition 3.1 is no longer true — the visible regions on a face of a 3D workpiece may be disjoint. To find the visible regions on a face, we can employ a spherical visibility map [Woo, 1994] or graphics-hardware based visibility algorithms [Khardekar et al., 2006]. Proposition 3.2 still holds true in 3D, therefore we only need to sample the boundary faces between the interference and cleaning C-spaces to find all the possible cleanable regions. In order to provide interactive 3D cleanability feedback to designers, however, the speed of prior 3D Minkowski sum algorithms would remain a bottleneck.

3.8 Summary

In this chapter we presented an approach for finding all the cleanable regions of a polygon under a 2D waterjet cleaning process by means of geometric accessibility analysis. Cleanability is determined by three factors — visibility from the nozzle, sufficient impact pressure, and a collision-free position of the lance. Our approach first constructs the cleaning C-space where the lance doesn't collide with the workpiece and at the same time the waterjet produces sufficient impact pressure for effective cleaning. Visibility testing is then performed to find the cleanable regions under a specific configuration. The boundary between the cleaning and interference C-spaces is sampled to find all the possible cleanable regions. Results demonstrate that our approach can be applied to different lance orientations and multi-orifice nozzles.

Chapter 4

Computing Voxelized 3D Minkowski Sums on the GPU

In section 1.2, we gave an example showing that the Minkowski sum $A \oplus B$ can be generated by sweeping A along the boundary of B (or vice versa). In this chapter, we will develop the mathematics behind the "sweep-along-the-boundary" behavior — we give both an accurate mathematical description and a strict proof (Proposition 4.1). Based on this mathematical formulation, we then introduce a new formula (Proposition 4.2) that decomposes the Minkowski sum of two closed watertight objects as the union of the Minkowski sum of their boundaries and a translation of each input object. Finally, for an efficient implementation, we use a stencil shadow volume technique to voxelize the union and create a solid voxelization of the Minkowski sum.

4.1 Mathematical Formulation

To manually draw the Minkowski sum of two simple shapes, one usually sweeps one shape along the boundary of the other. (In fact this method works for complex 3D objects too, but it is easier to perform manually for simple shapes.) Such an example is shown in Figure 4.1 (a). Here we sweep the yellow triangle along the boundary of the green rectangle. There are some subtleties that need to be clarified in this common approach, listed below and explained in Figure 4.1.

- A reference point needs to be specified for B. This reference point is placed on and swept along the boundary of A. Theoretically this point can be chosen arbitrarily. One usually uses a point on the boundary of B for convenience. In Figure 4.1 (a), we use point r as the reference point.

- Before B is swept, A needs to be translated by the vector defined by the reference point of B. Usually one simply sweeps B along the boundary of A without translating A, just because the origin is taken as the reference point. If the reference point is not the origin, a translation of A is necessary. In Figure 4.1 (a), the green square is translated by vector r.

- Just sweeping B along the boundary of A does not necessarily cover the whole Minkowski sum $A \oplus B$. As shown in Figure 4.1 (b), an empty rectangle inside the Minkowski sum is not covered by the sweep in this example. This empty rectangle will be covered by A after it is translated by vector r if r is chosen to be a point of B (either on the boundary or in the interior), as one normally does.

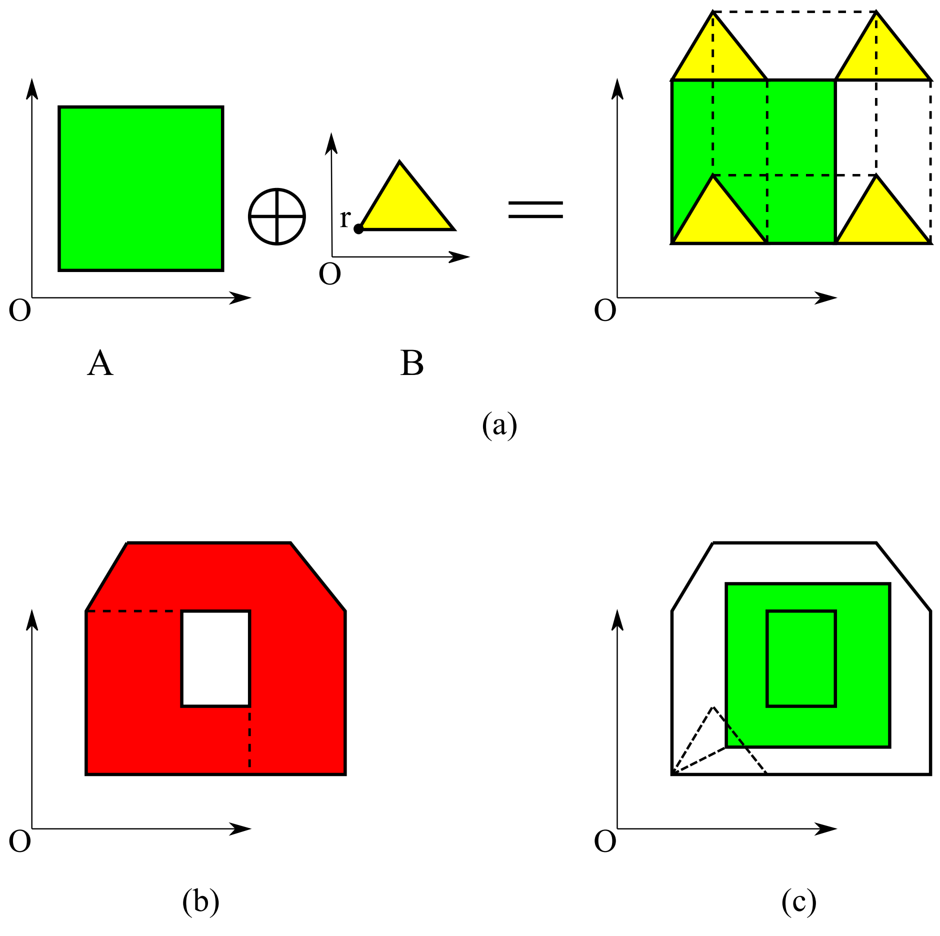


Figure 4.1: Minkowski sum $A \oplus B$ by sweeping B along the boundary of A. (a) Sweep B along the boundary of A. (b) An empty area inside the Minkowski sum is not swept by B. (c) Translation of A by any vector in B covers the empty area.

Mathematically "sweeping B along the boundary of A" is represented by $B \oplus \partial A$. To cover the empty area that is not covered by the sweep, we need a translation of A. A key observation is that as long as A is translated by a vector defined by *any* point of B, it will cover the empty area (see Figure 4.1 (c)). We summarize this observation in the proposition below and also give a strict proof. A similar proposition was introduced in the technical report of [Menon and Voelcker, 1993], though the origin was assumed to be inside of B (or translated before and after). In our proposition below, we do not put any restriction on the origin and give a more concise proof.

Proposition 4.1. *Suppose A and B are two singly connected watertight objects, and p_b is an arbitrary point of B (either on the boundary or in the interior). Then*
$A \oplus B = (B \oplus \partial A) \cup (A + p_b)$

Proof. Let C denote $(B \oplus \partial A) \cup (A + p_b)$, $\forall p_b \in B$. By definition, $B \oplus \partial A \subseteq A \oplus B$ and $A + p_b \subseteq A \oplus B$. Therefore $C \subseteq A \oplus B$. We only need to prove that $A \oplus B \subseteq C$; i.e., $\forall c \in A \oplus B$, we need to show that $c \in C$.

For any $c \in A \oplus B$ there must exist points a and b, $a \in A$ and $b \in B$, such that $c = a + b$. If $a \in \partial A$, then $c \in B \oplus \partial A \subseteq C$. If $a \notin \partial A$, then a must lie in the interior of A. Now we consider $-B + (a + b)$ (B is reflected around the origin and then translated by vector $a + b$). After this transformation, point b will coincide with point a (since $-b + (a + b) = a$). Note that b can be either on the boundary or in the interior of B. Now consider the intersection between the boundaries of A and $-B + (a + b)$. Since A and $-B + (a + b)$ share at least one common point a and a is in the interior of A, either these two boundaries intersect each other, or one of them completely contains the other, for a total of three different cases to consider (see Figure 4.2).

- case (1): $\partial(-B + (a + b))$ and ∂A intersect. Suppose a' is the intersection, $a' \in \partial A$ and $a' \in \partial(-B + (a + b))$. Since $a' \in \partial(-B + (a + b))$, $\exists b' \in B$ such that $a' = -b' + a + b$. Then $c = a + b = (a + b - b') + b' = a' + b' \in \partial A \oplus B \subseteq C$.

- case (2): $-B + (a + b) \subseteq A$. Then $\forall p_b \in B$, $-p_b + a + b \in A \Rightarrow -p_b + c \in A \Rightarrow c \in A + p_b \subseteq C$.

- case (3): $A \subseteq -B + (a + b)$. Then $\forall p_a \in \partial A \subseteq A$, $p_a \in -B + (a + b) \Rightarrow p_a \in -B + c \Rightarrow -p_a \in B + (-c) \Rightarrow c - p_a \in B \Rightarrow c \in B + p_a \subseteq B \oplus \partial A \subseteq C$.

Thus for all the three cases, we have $c \in C$. This proves $A \oplus B \subseteq C$. Thus $A \oplus B = C$.

□

Proposition 4.1 mathematically explains the "sweep-along-the-boundary" method used to generate Minkowski sums. In fact, as suggested by our proof above, we find that the proposition can be further extended. In case (1), b' must be on the boundary of B; in case (3), p_a can be any point of A, not just a point on the boundary of A. Another reason the proposition might be extended is that A and B are symmetric in $A \oplus B$ (i.e., they can be swapped without causing any change), but they are not in $(B \oplus \partial A) \cup (A + p_b)$.

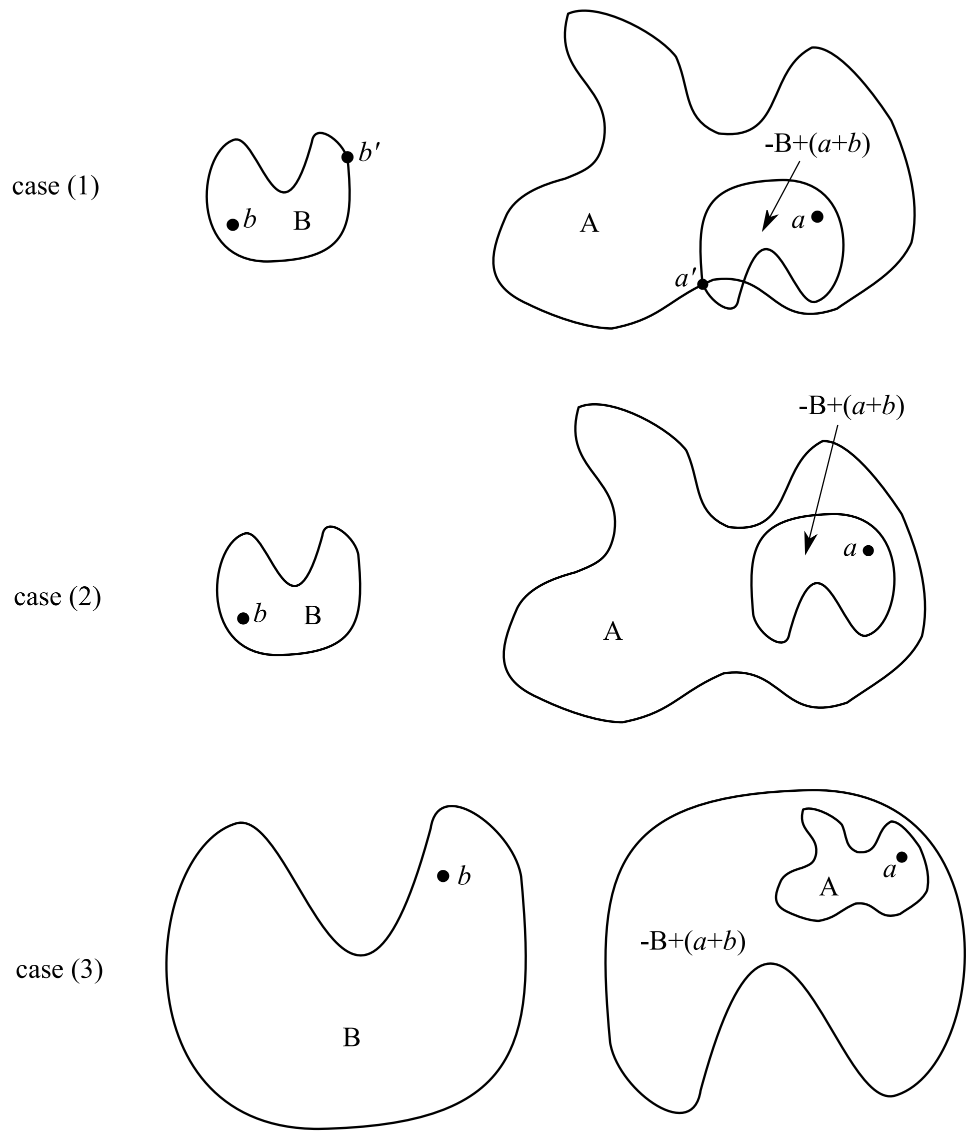


Figure 4.2: Proof of Proposition 4.1 and Proposition 4.2.

The extended proposition is given below. It shows that the Minkowski sum of two singly connected watertight objects can be decomposed as the union of the Minkowski sum of their boundaries and a copy of each object translated by a vector defined by *any* point of the other object. Its proof is very similar to the one for Proposition 4.1. To the best of our knowledge, this is the first mathematical formulation and explanation of the relationship between the Minkowski sum of two objects and the Minkowski sum of their boundaries.

Proposition 4.2. *Suppose A and B are two singly connected watertight objects, and p_a and p_b are two arbitrary points of A and B respectively (either on the boundary or in the interior). Then*
$A \oplus B = (\partial A \oplus \partial B) \cup (A + p_b) \cup (B + p_a)$.

Proof. Let C denote $(\partial A \oplus \partial B) \cup (A + p_b) \cup (B + p_a)$, $\forall p_a \in A$ and $\forall p_b \in B$. By definition, $\partial A \oplus \partial B \subseteq A \oplus B$, $A + p_b \subseteq A \oplus B$, and $B + p_a \subseteq A \oplus B$. Then $C \subseteq A \oplus B$. We only need to prove that $A \oplus B \subseteq C$; i.e., $\forall c \in A \oplus B$, we need to show that $c \in C$.

For any $c \in A \oplus B$ there must exist points a and b, $a \in A$ and $b \in B$, such that $c = a + b$. If $a \in \partial A$ and $b \in \partial B$, we have $c \in \partial A \oplus \partial B \subseteq C$. Otherwise, either $a \notin \partial A$ or $b \notin \partial B$. Without loss of generality we assume that $a \notin \partial A$. Then a must lie in the interior of A. Now we consider $-B+(a+b)$ (B is reflected around the origin and then translated by vector $a+b$). After this transformation, point b will coincide with point a (since $-b + (a + b) = a$). Note that b can be either on the boundary or in the interior of B. Now consider the intersection between the boundaries of A and $-B + (a + b)$. Since A and $-B + (a + b)$ share at least one common point a and a is in the interior of A, either these two boundaries intersect each other, or one of them completely contains the other, for a total of three different cases to consider (see Figure 4.2).

- case (1): $\partial(-B+(a+b))$ and ∂A intersect. Suppose a' is the intersection, $a' \in \partial A$ and $a' \in \partial(-B + (a + b))$. Since $a' \in \partial(-B + (a + b))$, $\exists b' \in \partial B$ such that $a' = -b' + a + b$. Then $c = a + b = (a + b - b') + b' = a' + b' \in \partial A \oplus \partial B \subseteq C$.

- case (2): $-B + (a + b) \subseteq A$. Then $\forall p_b \in B$, $-p_b + a + b \in A \Rightarrow -p_b + c \in A \Rightarrow c \in A + p_b \subseteq C$.

- case (3): $A \subseteq -B + (a + b)$. Then $\forall p_a \in A$, $p_a \in -B + (a + b) \Rightarrow p_a \in -B + c \Rightarrow -p_a \in B + (-c) \Rightarrow c - p_a \in B \Rightarrow c \in B + p_a \subseteq C$.

Thus for all the three cases, we have $c \in C$. This proves $A \oplus B \subseteq C$. Thus $A \oplus B = C$.

$\square$

Proposition 4.1 and 4.2 apply to any singly connected watertight object. Note that for a degenerate object that has zero volume (such as a surface, a line, or a point embedded in 3D space), its boundary is the same as the object itself. If both input models are triangulated polyhedra, their boundaries are the sets of their boundary triangles. Proposition 4.2 can then be rewritten as the corollary below.

Corollary 4.1. *Suppose A and B are two triangulated polyhedra (singly connected and watertight), F_A and F_B are the sets of their respective boundary triangles (faces), and p_a and p_b are two arbitrary points of A and B respectively (either on the boundary or in the interior), then we have*
$A \oplus B = (F_A \oplus F_B) \cup (A + p_b) \cup (B + p_a)$.

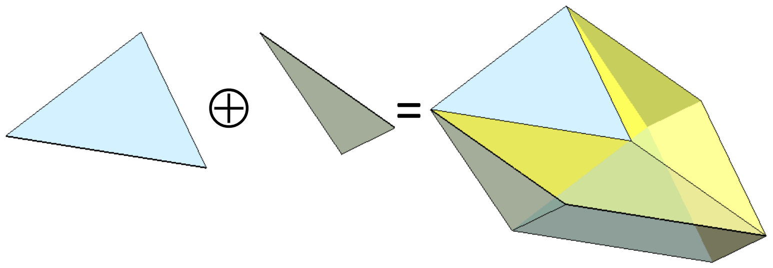


Figure 4.3: Minkowski sum of two triangles in 3D space.

To compute $F_A \oplus F_B$, we need to compute the Minkowski sum of two triangles in 3D space. Generally the Minkowski sum of two triangles in 3D (Figure 4.3) is a polyhedron with 9 boundary faces (4 triangles and 5 quadrilaterals). These boundary faces are determined by the relative position of the two triangles in 3D space. To further simplify the computation, we use the proposition below to reduce the Minkowski sum of two triangles to the union of Minkowski sums of triangles and edges, which are simply triangular prisms.

Proposition 4.3. *Suppose A and B are two triangles in 3D space, and E_A and E_B are the sets of their respective boundary edges, then*
$A \oplus B = (A \oplus E_B) \cup (B \oplus E_A)$.

Proof. Let C denote $(A \oplus E_B) \cup (B \oplus E_A)$. By definition, $A \oplus E_B \subseteq A \oplus B$ and $B \oplus E_A \subseteq A \oplus B$, so $C \subseteq A \oplus B$. It remains only to prove that $A \oplus B \subseteq C$.

If the two planes containing A and B respectively are not parallel, call their intersection line L (see Figure 4.4). If the two planes are parallel, take any line parallel to them as L. For any $c \in A \oplus B$ there must exist points a and b, $a \in A$ and $b \in B$, such that $c = a + b$. If at least one of a and b is on an edge of the two triangles, we have $c \in A \oplus E_B \subseteq C$ or $c \in B \oplus E_A \subseteq C$. Otherwise, both a and b are in the interior of the triangles; call the intersections of the line, passing through a (respectively, b) and parallel to L, with the edges of A (respectively, B) a_1 and a_2 (respectively, b_1 and b_2). We compare the lengths of the four line segments aa_1, aa_2, bb_1, and bb_2. Without loss of generality, suppose aa_1 has the smallest length out of the four, then $b + a - a_1 \in B$. This gives us $c = a + b = (b + a - a_1) + a_1 \in B \oplus E_A \subseteq C$. Note that the above reasoning still holds if all four line segments have the same length or if A and B intersect.

Therefore we have $\forall c \in A \oplus B$, $c \in C$; i.e., $A \oplus B \subseteq C$.

□

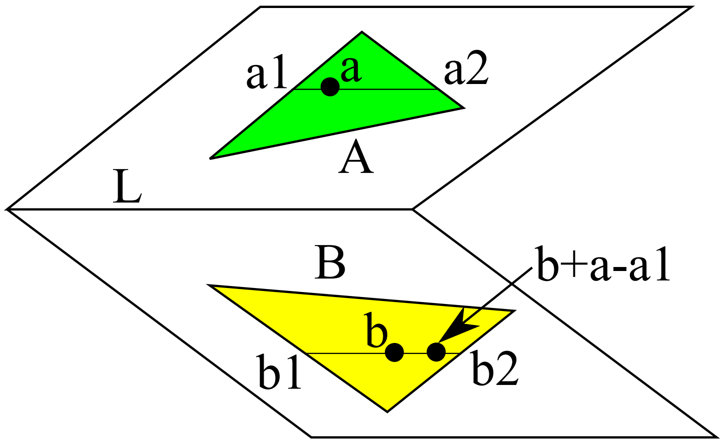


Figure 4.4: Proof of Proposition 4.3

Based on Corollary 4.1 and 4.3, we can decompose the Minkowski sum of two polyhedra as the union of a series of prisms and a translation of both input polyhedra, as shown in the proposition below.

Proposition 4.4. *Suppose A and B are two triangulated polyhedra, F_A and F_B are the sets of their respective boundary triangles, E_A and E_B are the sets of their respective edges, p_a and p_b are two arbitrary points of A and B respectively (either on the boundary or in the interior), then*
$A \oplus B = (F_A \oplus E_B) \cup (F_B \oplus E_A) \cup (A + p_b) \cup (B + p_a)$.

Proof. For any triangle $f_A \in F_A$ and $f_B \in F_B$, suppose the edge sets of f_A and f_B are E_{f_A} and E_{f_B} respectively. From Proposition 4.3, $f_A \oplus f_B = (f_A \oplus E_{f_B}) \cup (f_B \oplus E_{f_A})$. Then from Proposition 2.5 (distributivity), we have

$$\begin{aligned}
F_A \oplus F_B &= (\cup_{f_A \in F_A} \{f_A\}) \oplus (\cup_{f_B \in F_B} \{f_B\}) \\
&= \cup_{f_A \in F_A, f_B \in F_B} (f_A \oplus f_B) \\
&= \cup_{f_A \in F_A, f_B \in F_B} ((f_A \oplus E_{f_B}) \cup (f_B \oplus E_{f_A})) \\
&= (\cup_{f_A \in F_A, f_B \in F_B} f_A \oplus E_{f_B}) \cup (\cup_{f_A \in F_A, f_B \in F_B} f_B \oplus E_{f_A}) \\
&= ((\cup_{f_A \in F_A} \{f_A\}) \oplus (\cup_{f_B \in F_B} E_{f_B})) \cup ((\cup_{f_B \in F_B} \{f_B\}) \oplus (\cup_{f_A \in F_A} E_{f_A})) \\
&= (F_A \oplus E_B) \cup (F_B \oplus E_A) \, .
\end{aligned}$$

Now applying Corollary 4.1,

$$\begin{aligned}
A \oplus B &= (F_A \oplus F_B) \cup (A + p_b) \cup (B + p_a) \\
&= (F_A \oplus E_B) \cup (F_B \oplus E_A) \cup (A + p_b) \cup (B + p_a) \, .
\end{aligned}$$

□

From Proposition 4.4, we know that the Minkowski sum of two polyhedra can be decomposed as the union of the following four components:

- $F_A \oplus E_B$, which is a series of prisms formed by sweeping each triangle of A along each edge of B;
- $F_B \oplus E_A$, which is a series of prisms formed by sweeping each triangle of B along each edge of A;
- $A + p_b$, which is a translation of A by a vector defined by any point of B;
- $B + p_a$, which is a translation of B by a vector defined by any point of A.

Each one of these four components can be computed easily. To compute Minkowski sums using Proposition 4.4, we need to find a way to compute their union efficiently, which will be described in the next section.

4.2 Voxelization Using Stencil Shadow Volumes

Suppose object A has $|F_A|$ triangles and $|E_A|$ edges, and object B has $|F_B|$ triangles and $|E_B|$ edges, then in total we have $|F_A| \cdot |E_B| + |F_B| \cdot |E_A|$ prisms for $(F_A \oplus E_B) \cup (F_B \oplus E_A)$. Including the two translated objects $A+p_b$ and $B+p_a$, we need to compute the union of $|F_A| \cdot |E_B| + |F_B| \cdot |E_A| + 2$ polyhedral objects in order to compute $A \oplus B$ using Proposition 4.4. The combinatorial complexity of the union can be as high as $O(|F_A| \cdot |E_B| + |F_B| \cdot |E_A| + 2)^3$ [Aronov et al., 1997], which is of the same magnitude as the worst-case complexity $O(|F_A|^3 \, |F_B|^3)$ of the Minkowski sum $A \oplus B$. Even if both A and B have just thousands of triangles, the union consists of millions of objects, and in the worst case its number of facets can be on the order of 10^{18}. Computing an exact union of such a large number of polyhedra is not practical. Exact boundary evaluation of a union of this size is slow and prone to robustness problems.

Instead of computing the union exactly, we propose approximating it by using GPU-based voxelization techniques. In a voxelized representation, each object M is represented by a 3D grid M_{ij} ($1 \leq i,\ j \leq n$, where n is the resolution) that corresponds to the 3D space occupied by a bounding box enclosing the object. Each voxel M_{ij} is set to either 1 or 0 according to whether it is located inside or outside of the object. In a conservative voxelization [Zhang et al., 2007], as long as the voxel intersects with the object, it is set to 1 (Figure 4.5 (a)); in other voxelizations, the center of the voxel is used as a representative point to determine whether the voxel is inside or outside of the object (Figure 4.5 (b)), which is also the behavior of standard rasterization algorithms required by 3D graphics APIs such as OpenGL [OpenGL, 2011] and Direct3D [Direct3D, 2011]. In our approach for voxelizing Minkowski sums, we follow the latter convention.

Voxelization algorithms can be classified into surface voxelization (only the boundary is voxelized) and solid voxelization (the whole interior is voxelized). A surface voxelization can

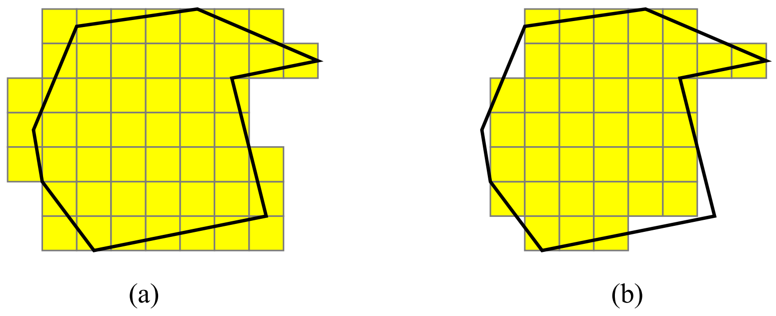


Figure 4.5: Conservative (a) and non-conservative (b) voxelization

be created from a solid voxelization by simply checking if a voxel has any outside neighbors (this technique will be demonstrated in the next chapter). A solid voxelization can also be created from a surface voxelization by using the parity check rule (explained below), if a single watertight object is being voxelized [Fang and Chen, 2000]. In this chapter we discuss algorithms for directly creating a solid voxelization for the Minkowski sum, since our goal is to avoid the computational complexity involved in evaluating the complete boundary of the Minkowski sum.

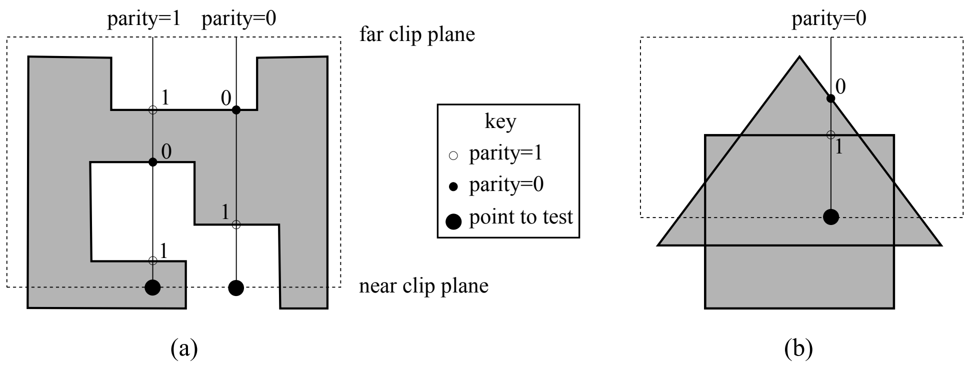


Figure 4.6: (a) Parity check for a point outside the object and a point inside the object. (b) The parity check does not work for the union of several watertight objects.

Most solid voxelization algorithms are based on parity checking and work for a single watertight object [Fang and Chen, 2000, Dong et al., 2004, Eisemann and Décoret, 2008, Heidelberger et al., 2003, Liao, 2008, Nooruddin and Turk, 2003]. The parity check is based on the principal that a ray shooting from a point inside (or outside) the object will have an odd (or even) number of intersections with the object boundary respectively, as shown in Figure 4.6 (a). To perform the parity check using OpenGL, we can set the near clipping plane to the current slice (all the pixels on this slice will be checked together) and the far clipping plane to be infinity, and then render the object. Only the portion of the object that is between the near and far clipping planes is rendered. The parity flag for each pixel is initialized to be zero. Then it is toggled between one and zero for each rendered fragment at its position (these fragments can be processed in arbitrary order). After the rendering is complete, any pixel inside the object will have a flag of one, and any pixel outside will have a flag of zero. Flag toggling can be implemented using either XOR operations with the color buffer or GL_INVERT operations with the stencil buffer.

The parity check, however, does not work for the *union* of watertight objects. As shown in Figure 4.6 (b), to voxelize the union of a rectangle and a triangle, the parity flag of the indicated point is zero after rendering, but in fact it is inside the union. We need a slightly more complex check to correctly classify inside and outside voxels for the union of objects.

The technique used in stencil shadow volumes [Everitt and Kilgard, 2002, McGuire et al., 2003] has been applied to solid voxelization of individual watertight objects as well as their union [Llamas, 2007, Liao and Fang, 2002]. It strengthens the above parity check by taking into consideration the orientation of boundary surfaces. Instead of simply toggling the flag between one and zero for each rendered fragment, they increase it by one for back faces and decrease it by one for front faces, as shown in Figure 4.7. After the rendering is complete, any pixel inside the union will have a non-zero stencil value, while any pixel outside the union will have a zero stencil value. The increments and decrements can be implemented using the two-sided stencil test provided by OpenGL extension GL_EXT_stencil_two_side. Note that the increments and decrements are performed with wrapping enabled to avoid saturation (otherwise the stencil value will be clamped at the maximum value and zero); thus for an 8-bit stencil buffer, increasing 255 by one will return 0 and decreasing 0 by one will return 255.

The final voxelization is stored in a 3D texture, where a 3D texture of size $X \times Y \times Z$ can be seen as Z images of size $X \times Y$ stacked together, each one of which can be independently set as the draw buffer to store the color of currently rendered pixels. Each image is a 2D array of texels (texture elements), and each texel has four color channels (RGBA), with each color channel using eight bits. Thus each image has 32 slices, with each bit representing the plane of a specific slice. Each voxel is represented by a single bit, thus for a voxelization with a resolution of $512 \times 512 \times 512$, the 3D texture should have a size of $512 \times 512 \times 16$ $(16 = 512/32)$, which uses only 16 MB of video memory.

We map bits of the 3D texture to the voxels of the voxelization using color encoding [Dong et al., 2004]. Suppose the resolution of the voxelization is $N \times N \times N$, the index of a voxel is (x_v, y_v, z_v) (ranging from 0 to $N-1$), and it corresponds to the n_{bit} bit of the texel with index (x_t, y_t, z_t) in the 3D texture. Then the following relationships holds:

$$\begin{aligned} x_t &= x_v \\ y_t &= y_v \\ z_t &= \lfloor z_v/32 \rfloor \\ n_{bit} &= z_v - 32z_t. \end{aligned} \tag{4.1}$$

Figure 4.8 shows a 2D example of color encoding. For the purpose of illustration, we assume

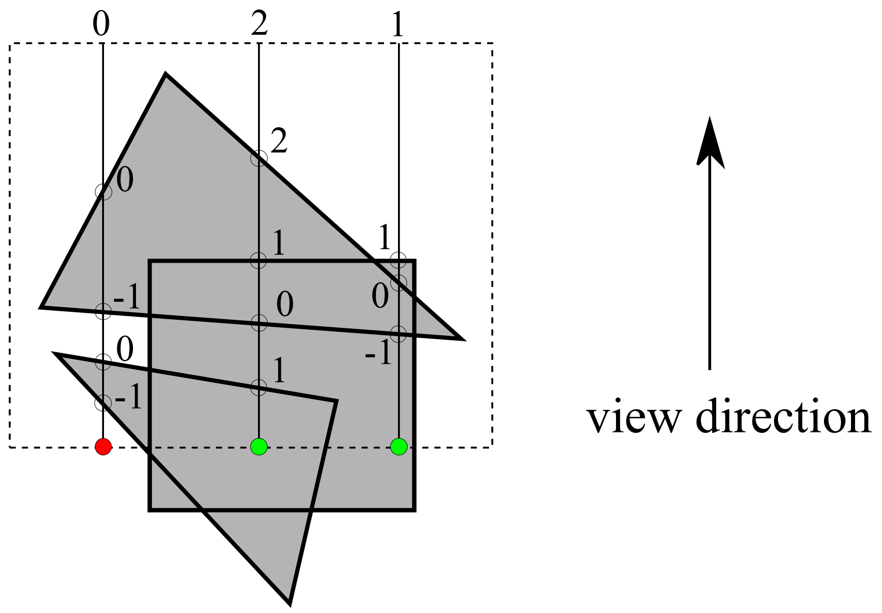


Figure 4.7: The stencil shadow volume technique. The stencil value is increased for back faces and decreased for front faces.

each texel has a bit depth of eight, though in reality it is 32.

4.3 Algorithm and Implementation

The overall algorithm for computing a solid voxelization of $A \oplus B$ for closed watertight objects A and B is given in Algorithm 4.1 and explained below. Suppose the resolution of voxelization is $N \times N \times N$. We first need to create a 3D texture of size $N \times N \times (N/32)$ with a bit depth of 32. We also create a framebuffer object and a stencil buffer, and attach the stencil buffer to the framebuffer object. Next we create a display list for all the triangular prisms and two translated objects according to Proposition 4.4. The bounding box of the Minkowski sum, which can be computed easily by adding up the minimum and maximum points of the two input models along the x, y, and z directions, is divided into N slices along the z direction.

Now we perform the voxelization slice by slice. For each slice, we first need to set the near clipping plane at the current slice and the far clipping plane slightly beyond the maximum z of the bounding box. We also need to set the two-sided stencil operation to be increasing for back faces and decreasing for front faces. We disable the draw buffer for now since we do not need to render the display list to a color buffer; instead we only need to fill the stencil buffer. Now we call the display list to render the prisms and the two translated objects. Note that

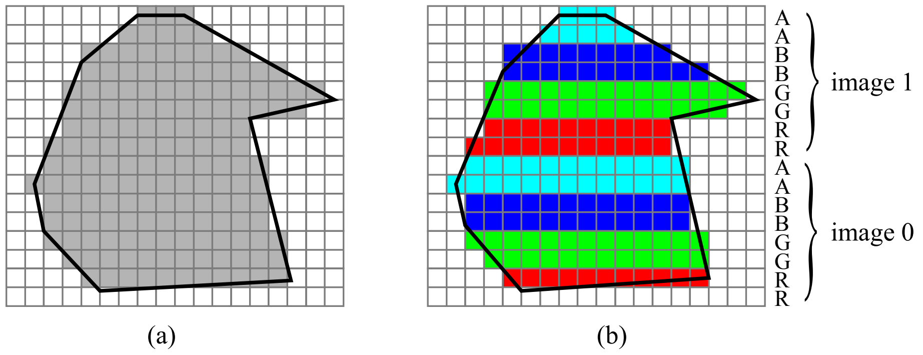


Figure 4.8: Color encoding of the voxels (2D illustration). (a) The voxels of a solid voxelization. (b) The corresponding texture (each texel has a bit depth of eight, two for each color channel). It includes two images (16 slices). The colored bits are set to 1 and the others are 0.

only the portion between the near and far clipping planes are rendered. After rendering, the stencil buffer is filled with zeros and non-zeros, indicating whether the corresponding voxel is outside or inside the Minkowski sum.

Now we can use the stencil buffer to find corresponding bits in the 3D texture. To do this, we first need to attach the current image of the 3D texture (we have $N/32$ images in total, and each image corresponds to 32 slices) as the draw buffer. If it is the i^{th} slice ($0 \leq i \leq N-1$), we attach the $\lfloor i/32 \rfloor^{th}$ image, so we only need to change the attached image every 32 slices. We also need to compute an RGBA color in which only the bit corresponding to the current slice is set to one and all the other bits are zeros. In fact we create a table in advance that includes all the 32 possible RGBA color values (0x00000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010, ..., 0x40000000, and 0x80000000), and look up the correct color value in the table instead of computing it on the CPU. For the i^{th} slice, we use the $(i - 32\lfloor i/32 \rfloor)^{th}$ color value in the table. The logical pixel operation is set to OR to avoid overwriting previously computed bits. Since we only need to write to the pixels whose stencil value is nonzero, we set the stencil test to pass if the stencil value is nonzero. Now we render a quadrilateral that is slightly larger than the xy projection of the bounding box to set the bits on the current slice.

We repeat this process for all the slices. Finally the 3D texture will contain all the information of the solid voxelization.

The main drawback of the above algorithm is the large number of prisms to be rendered. If both A and B have thousands of triangles, there will be millions of prisms. However, we can show that we do not need to render all the prisms for each slice. If a prism does not intersect with the current slice, it is either entirely between the near and far clipping planes

Algorithm 4.1 Compute a solid voxelization of resolution $N \times N \times N$ for $A \oplus B$

```
1: Create a 3D texture, a stencil buffer, and a framebuffer object.
2: Create a display list of all the triangular prisms and the two translated objects.
3: for each slice do
4:     Set up the view frustum.
5:     Set up the two-sided stencil operation.
6:     Disable the draw buffer.
7:     Call the display list to render to the stencil buffer.
8:     Attach the corresponding image of the 3D texture as the draw buffer.
9:     Compute the RGBA color corresponding to the current slice.
10:    Set the logical pixel operation to OR.
11:    Set the stencil test such that it passes if the stencil value is not zero.
12:    Render a quadrilateral with the computed color.
13: end for
14: return the 3D texture.
```

(prism C in Figure 4.9), which does not change the final stencil value, or entirely in front of the near clipping plane (prism B in Figure 4.9), which is not rendered at all. Therefore for each slice, instead of rendering all the prisms, we only need to render those prisms with which the current slice intersects. Especially for complex polyhedra generated by tessellating smooth models, their boundary triangles are usually very small. The prisms formed by these triangles are therefore small compared to the Minkowski sum, and they usually intersect with only a small number of slices. Then by first computing a list of prisms intersecting with each slice, we can reduce the number of prisms rendered for each slice and therefore the time needed for rendering. It is straightforward to check whether a prism intersects with a slice or not — since the slice is perpendicular with the z axis, we just need to check whether its z value is between the minimum and maximum z coordinates of the prism.

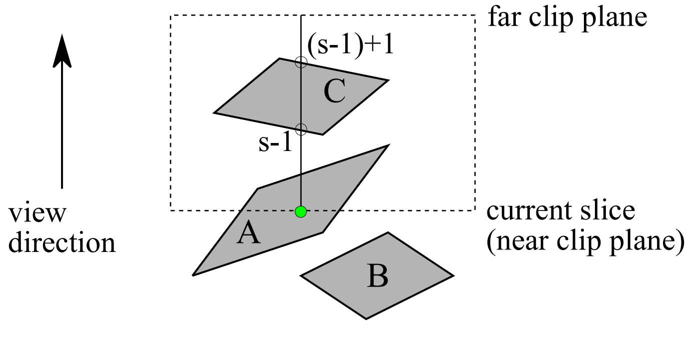


Figure 4.9: A prism only needs to be rendered for the slices that it intersects with. For the current slice, prism B and C do not need to be rendered.

On the other hand, finding intersecting prisms for each slice also incurs some overhead. If we use just a single display list for all the slices, we only need to create and evaluate (i.e., process the input draw commands to generate final pixel information) the display list once, and then reuse it repeatedly without re-evaluating the data over and over again. But if we render different sets of prisms for each slice, we lose the benefits from using a display list; i.e., we have to re-evaluate the data for each slice.

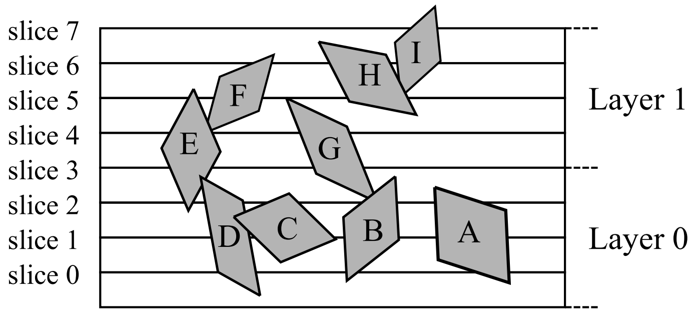


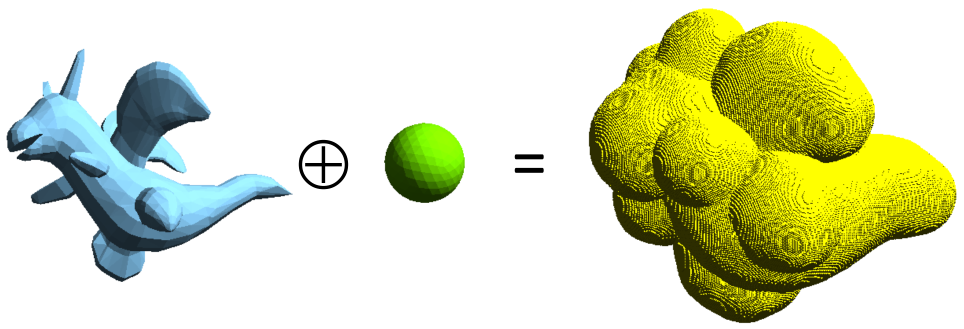
Figure 4.10: Slices are grouped into layers. In this example, layer 0 includes slice 0 to 3, and layer 1 includes slice 4 to 7. The intersecting prisms for layer 0 are $\{A, B, C, D, E, G\}$, and for layer 1 are $\{E, F, G, H, I\}$.

Therefore as a tradeoff, we group all the slices into several layers, each layer including a fixed number of slices. For each layer we compute a list of prisms intersecting with it, as shown in Figure 4.10. We render the same list of prisms for all the slices in one layer, so they can reuse a single display list.

Two factors need to be considered for choosing an appropriate value for the number of layers — the number and the size of display lists. If the number of layers is small, the generated display lists for each layer will be large; if the number of layers is large, the display lists will have smaller sizes, but at the same time we need to create more display lists. To determine an optimal value for the number of layers, we compute the voxelization of two Minkowski sums, representing two different types of input, using varying numbers of layers and compare the running timing. The first example (dragon $\oplus$ ball, Figure 4.11) uses two polyhedra generated by tesselating smooth models, and the second (grate1 $\oplus$ grate2, Figure 4.12) uses two rectilinear models. The timings of both examples show that for both resolutions of $256 \times 256 \times 256$ and $512 \times 512 \times 512$, using 16 layers has the best performance. Compared to using a single display list for all the slices (i.e., the number of layers is one), using 16 layers has a $3\times$ to $6\times$ speedup.

4.4 Results and Discussion

In this section, we give some results of the above voxelization algorithm and also their timings. The test models we use come from Lien's website [Lien, 2011] and were also used



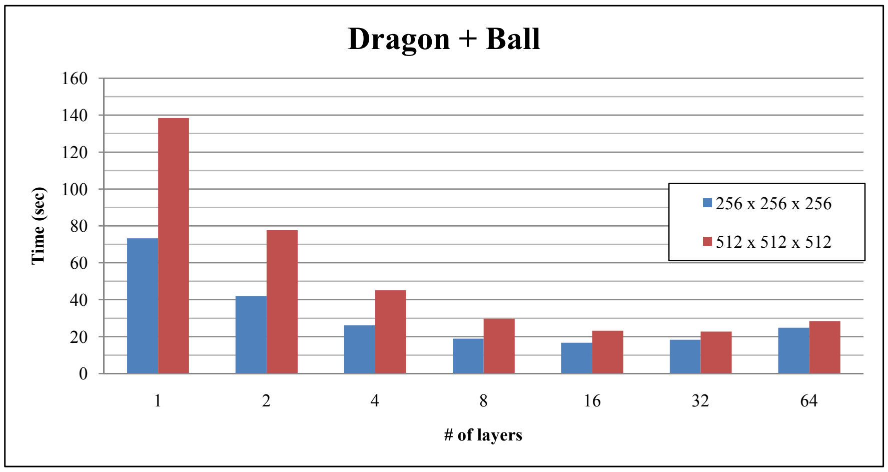
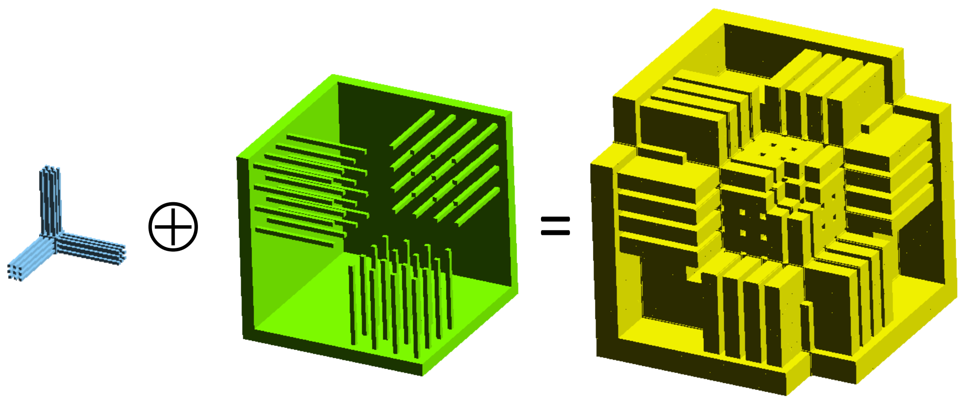


Figure 4.11: Running time for computing the voxelization of the Minkowski sum of the dragon (2328 triangles) and the ball (500 triangles) using different numbers of layers. Both resolutions of $256 \times 256 \times 256$ and $512 \times 512 \times 512$ are used for comparison. The Minkowski sum on the top row is rendered using resolution $256 \times 256 \times 256$.

in [Lien, 2008d]. The timing was performed on an NVIDIA Quadro 6000 GPU with 6 GB video memory and an Intel Core 2 Quad CPU at 2.66 GHz with 4 GB RAM. The program runs on CUDA driver 3.2 and 64-bit Windows 7. For all the tests we divide the slices into 16 layers.

The timings of the voxelization algorithm are given in Table 4.1. We also include the timings reported by Lien in [Lien, 2008d] for comparison (performed on two Intel Core 2 CPUs at 2.13 GHz with 4 GB RAM). Note that Lien's approach uses a convolution-based approach and computes a boundary representation, while ours computes a voxelized representation. The final voxelized Minkowski sums are shown in Figure 4.13, except "grate1 $\oplus$ grate2", which was already shown in Figure 4.12.

From Table 4.1, we can see that the performance of our algorithm is mainly dominated by the sizes (numbers of triangles) of the input polyhedra, while the performance of Lien's approach is determined by both the sizes and the shape complexity of the input models.



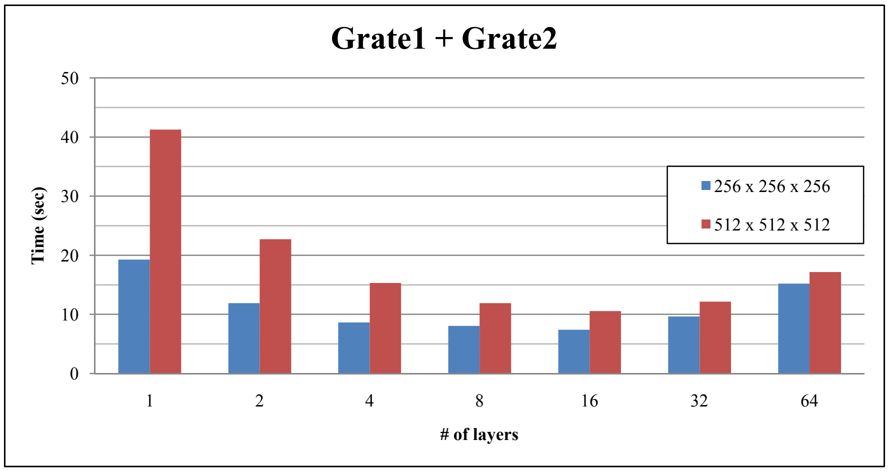


Figure 4.12: Running time for computing the voxelization of the Minkowski sum of the grate1 (540 triangles) and the grate2 (942 triangles) using different numbers of layers. Both resolutions of $256 \times 256 \times 256$ and $512 \times 512 \times 512$ are used for comparison. The Minkowski sum on the top row is rendered using resolution $256 \times 256 \times 256$.

This is especially obvious for the "grate1 $\oplus$ grate2" case, which is a worst case example for concave Minkowski sum computation with $O(m^3n^3)$ faces [Varadhan and Manocha, 2006]. Even though the two input polyhedra do not have large numbers of triangles compared to the other examples, Lien's approach takes 318 seconds to compute their Minkowski sum boundary, while our approach needs just 10 seconds to compute a $512\times512\times512$ voxelization. Another example is the "clutch $\oplus$ ball" case. Since one of the input models is convex, Lien's approach can take advantage of the convexity so that it takes less than 3 seconds to compute the Minkowski sum, but our algorithm takes around 20 seconds to compute a $256\times256\times256$ voxelization.

One advantage of the voxelization algorithm described in this chapter is ease of implementation. It does not require the complex 3D Boolean operations that are usually involved in existing B-rep based algorithms — the only necessary 3D computation is computing the

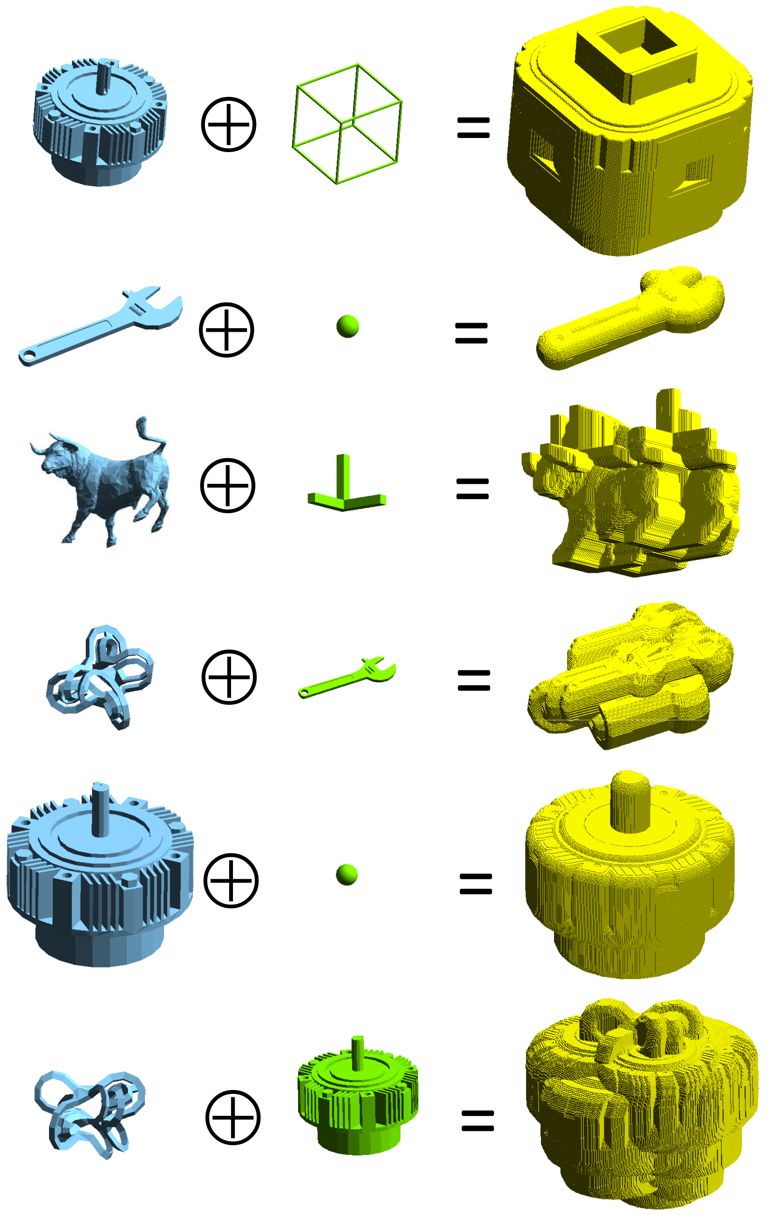


Figure 4.13: Voxelization results of the other examples given in Table 4.1. The voxelizaition is computed using a resolution of $256 \times 256 \times 256$.

A	B	#triA	#triB	512^3	256^3	Lien
grate1	grate2	540	942	9.56	6.65	318.50
clutch	frame	2,116	96	6.26	4.27	23.50
wrench	ball	772	500	11.26	8.10	0.90
bull	axes	12,396	36	10.80	7.82	22.10
knot	wrench	992	772	18.60	13.17	37.00
clutch	ball	2,116	500	26.34	19.20	2.70
knot	clutch	992	2,116	49.60	36.23	347

Table 4.1: Timings of the voxelization algorithm (in seconds). Two resolutions ($512 \times 512 \times 512$ and $256 \times 256 \times 256$) are used for testing. The timings reported by Lien in [Lien, 2008d] are also included for comparison.

triangular prism formed by a triangle and an edge, which is straightforward. However, its main drawback is that the performance does not improve significantly on prior results, which has two explanations. The first is that a large number of prisms need to be rendered even if the input models have a moderate size (thousands of triangles). For most of the examples given in Table 4.1, we need to render millions of prisms. The other is that our algorithm does not consider the specific shapes of the input models. If one of the input models is convex, its convexity can be used to simplify the computation of Minkowski sums [Barki et al., 2009a, Varadhan and Manocha, 2006, Lien, 2008d], but it is not utilized in this voxelization algorithm. (This explains why our algorithm takes much longer than Lien's to compute "clutch $\oplus$ ball" in Table 4.1). In the next chapter, we will present another voxelization algorithm that overcomes these two shortcomings and has significantly improved performance, at the expense of not computing inner voids that may exist in the Minkowski sum.

One limitation of this approach is that it is not robust in the presence of floating point rounding errors. When we compute the triangular prism formed by a triangle abc and an edge de, we need to determine the correct orientation of each boundary face of the prism, since the orientation determines whether the boundary face is a front face or back face. As shown in Figure 4.14, there are two possible orientations for the boundary triangle formed by the three vertices $a + e$, $b + e$, and $c + e$. In Figure 4.14 (a), the triangle is oriented as $a + e \to b + e \to c + e$; while in Figure 4.14 (b), it is oriented as $a + e \to c + e \to b + e$. Computing the orientation of a face suffers from floating point errors if the volume of the prism is very close to zero. If the orientation of the boundary faces of a prism is wrongly computed due to rounding errors, it may cause a hole inside the Minkowski sum. But these holes are very small since the volume of a prism with wrongly oriented faces is close to zero.

To compute an accurate orientation, one needs to use exact arithmetic [Shewchuk, 1997, Halperin, 2002]. Implementing such exact arithmetic is not trivial and it will unavoidably impact the performance. In the voxelization algorithm to be discussed in the next chapter, we use a conservative method to adaptively bound floating point errors and avoid computing

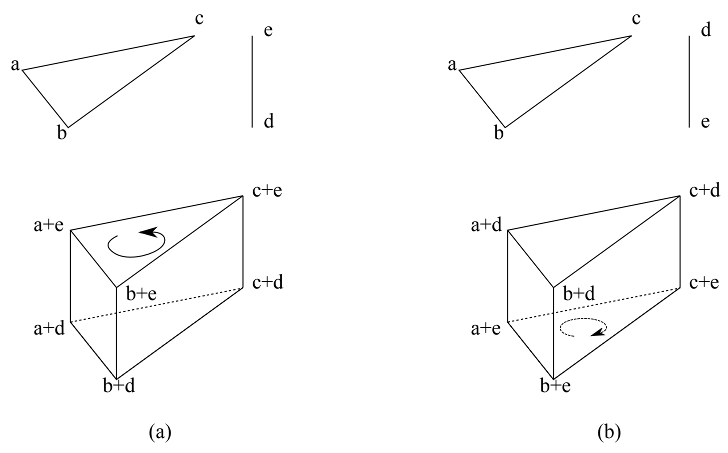


Figure 4.14: Two possible orientations for a boundary face of the prism formed by a triangle *abc* and an edge *de*.

exact orientations.

Chapter 5

Computing Voxelized 3D Minkowski Sums Without Holes

5.1 Introduction

In this chapter we present a new approach for computing Minkowski sums (excluding any enclosed voids) of arbitrary polyhedra that extends and improves upon previous convolution-based algorithms [Kaul and Rossignac, 1992, Lien, 2008d] and combines these methods with GPU-based voxelization techniques. Similar to the algorithm introduced in the previous chapter, the new algorithm aims to directly create a voxelization of the Minkowski sum, without having to compute a complete boundary representation. Meanwhile we provide a boundary visualization for display. The volumetric data is stored and computed exclusively on the GPU to utilize its rasterization functionality and parallel computation capacity.

Our approach consists of two main steps — primitive culling (section 5.2) and voxelization (section 5.3). We first cull out surface primitives that will not contribute to the final boundary of the Minkowski sum, analyzing and adaptively bounding the rounding errors of the culling algorithm to solve the floating point error problem. The remaining surface primitives are then rendered to depth textures along six orthogonal directions to generate an initial solid voxelization of the Minkowski sum. Then we employ fast flood fill to find all the outside voxels. We generate both solid and surface voxelizations of Minkowski sums without enclosed voids and support high volumetric resolution of 1024^3 with low video memory cost.

The benefits of this voxelization algorithm include:

- *easy implementation*: Our approach avoids the complex 3D computations involved in convex-decomposition and convolution approaches.

- *high speed*: Our algorithm is at least one order of magnitude faster than existing B-rep based algorithms.

- *memory efficiency*: Our voxelized Minkowski sum only requires 128MB video memory for a resolution of 1024^3.
- *multiresolution*: Users can choose different volumetric resolutions according to the tolerance requirements of different applications.
- *robustness*: We analyze floating point rounding errors of the culling algorithm to ensure correct voxelization.

The accuracy of our algorithm is governed by the volumetric resolution. Since we support a relatively high resolution of 1024^3 by using volume encoding (section 5.3.1), we can achieve an accuracy of 0.085% (measured by the minimum distance from centers of boundary voxels to the actual Minkowski sum boundary, $\sqrt{3}/2/1024$, or $\sqrt{3}/2/N$ for a resolution of N^3), which is enough for most applications.

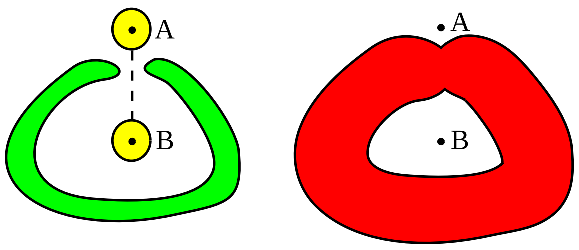


Figure 5.1: The 2D Minkowski sum (on right in red) of a yellow disk and a green belt contains an enclosed void. The yellow disk can be placed at B, but it cannot go from A to B.

One limitation of this work is that we do not compute enclosed voids of a Minkowski sum, i.e., we only identify its outer boundary. Usually in motion planning, we do not need to consider enclosed voids in Minkowski sums, because they represent locations where the object can be placed without collision, but cannot be reached from the outside (Figure 5.1).

5.2 Rendering the Outer Boundary of Minkowski Sums

In this section we introduce a GPU-based algorithm for rendering the outer boundaries of Minkowski sums, without having to compute a correct and complete boundary representation. The voxelization algorithm presented in 5.3 is built upon the rendering results. The rendering algorithm described here can also be used as a stand-alone visualization system for 3D Minkowski sums. It can also be directly applied to implement polyhedron interpolation and morphing.

We first introduce the terminology used in this section. We assume the two input polyhedra A and B are 2-manifold triangular meshes. Let $F_A = \{f_A\}$ and $F_B = \{f_B\}$ be the boundary triangle sets, $E_A = \{e_A\}$ and $E_B = \{e_B\}$ be the edge sets, and $V_A = \{v_A\}$ and $V_B = \{v_B\}$ be the vertex sets of A and B respectively.

The rendering algorithm first computes a set of surface primitives that is a superset of the Minkowski sum boundary. Surface primitives that do not contribute to the boundary are culled out in parallel on the GPU. The remaining primitives are written to a VBO (Vertex Buffer Object), which is then rendered directly using OpenGL.

5.2.1 Surface Primitive Culling

It has been shown in [Kaul and Rossignac, 1992] that any facet on the boundary of $A \oplus B$ is generated in one of the following three ways: translating a triangle in F_A by a vector in V_B, translating a triangle in F_B by a vector in V_A, or sweeping an edge in E_A along an edge in E_B. We call the triangles formed by the first two methods *triangle primitives*, and the quadrilaterals formed by the third method *quadrilateral primitives*.

The counts of all the triangle and quadrilateral primitives are $|F_A| \cdot |V_B| + |V_A| \cdot |F_B|$ and $|E_A| \cdot |E_B|$ respectively. These numbers are as high as millions for two polyhedra with thousands of triangles, similar to the previous chapter. It will take a large amount of time and memory to render all these primitives. For example, 1 GB of video memory will limit the size of A and B to just a few thousands of triangles. Note, however, that a large number of surface primitives lie entirely inside the Minkowski sum and will be hidden during the rendering (see Figure 5.2 for a 2D example). We can cull out these primitives and render only the remaining ones. As to be shown in Table 5.1, this will greatly reduce the number of primitives to be rendered. In addition to these completely hidden primitives, some primitives are trimmed by others and become partially hidden during the rendering (also shown in Figure 5.2). Convolution-based algorithms for computing the Minkowski sum boundary identify and compute all such intersections, but for the purpose of rendering, we allow them to be handled automatically in the graphics pipeline by using the appropriate depth test.

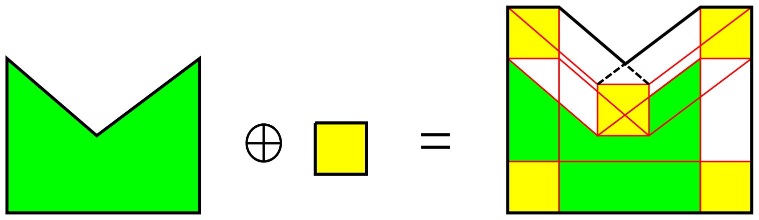


Figure 5.2: A 2D example of surface primitives in the interior of the Minkowski sum (shown as red lines) and trimmed surface primitives (shown as dashed lines).

In the text that follows we call a surface primitive *contributing* if its intersection with the Minkowski sum boundary has a non-zero area (so a partially trimmed primitive is contributing since it has a partial overlap with the Minkowski sum boundary); otherwise we call it *noncontributing*. As the name implies, contributing primitives will "contribute" to the boundary of the Minkowski sum, but noncontributing ones will not. Note that according to our definition, a surface primitive that only shares an edge or vertex with the Minkowski sum boundary is noncontributing, because their intersection has an area of zero.

The rendering algorithm developed in this chapter is based on several propositions for primitive culling. The first two propositions (Proposition 5.1 and 5.2 below) were first introduced in [Kaul and Rossignac, 1992]. However, no proof was provided in that paper. These two propositions were used later, also unproved, in other works [Lien, 2008a,d]. In [Liu et al., 2009] the authors proved similar propositions, but their criterion for triangle primitive culling is weaker than the one stated below (Proposition 5.1). Here we use a mathematical description and give proofs of these two propositions.

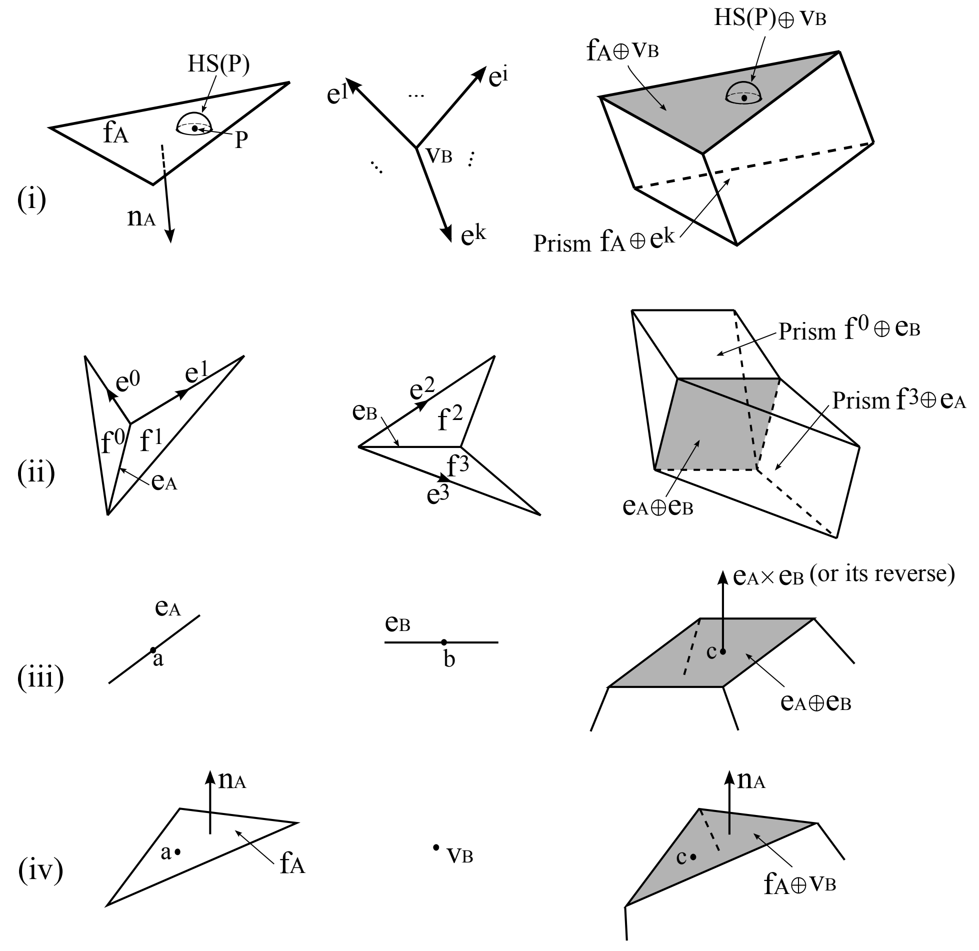


Figure 5.3: Illustration of the proof of Proposition 5.1 (i), 5.2 (ii), 5.3 (iii), and 5.4 (iv).

Proposition 5.1. *Given $f_A \in F_A$ and $v_B \in V_B$, with n_A the outward facing normal of f_A, and e^i the i^{th} incident edge pointing away from v_B. If $f_A \oplus v_B$ is a contributing triangle primitive, then $n_A \cdot e^i \leq 0$, $\forall e^i$.*

Proof. (By contradiction.) Suppose $\exists e^k$ such that $n_A \cdot e^k > 0$ (Figure 5.3 (i)). Since A is a 2-manifold, for any point P inside the triangle f_A, we can find a hemisphere $HS(P)$ with a small radius r, centered at P and entirely inside A, i.e.,

$$\begin{aligned} HS(P) &= \{Q : \|Q-P\| \leq r, (Q-P)\cdot n_A \leq 0\} \\ HS(P) &\subseteq A. \end{aligned}$$

Then we consider the translated hemisphere $HS(P) \oplus v_B$ and the prism generated by $f_A \oplus e^k$. They locate on different sides of the triangle $f_A \oplus v_B$ (shaded in the figure). Since P is inside f_A, we can always reduce the radius of $HS(P)$ such that the other half of the hemisphere $HS(P) \oplus v_B$ is entirely inside the prism $f_A \oplus e^k$. This means that for each point inside the triangle $f_A \oplus v_B$, we can always find a small sphere around it and the sphere is a subset of $A \oplus B$ (remember that $HS(P) \oplus v_B \subseteq A \oplus v_B \subseteq A \oplus B$ and $f_A \oplus e^k \subseteq A \oplus B$). So $f_A \oplus v_B$ will not overlap with the boundary of $A \oplus B$. This contradicts the assumption that $f_A \oplus v_B$ is contributing. □

Proposition 5.2. *Suppose $e_A \in E_A$ and $e_B \in E_B$, f^0 and f^1 are the two incident triangles of e_A, and e^0 (or e^1) is one of the two edges of f^0 (or f^1) pointing away from e_A. Let f^2, f^3, e^2 and e^3 be defined similarly for e_B. If $e_A \oplus e_B$ is a contributing quadrilateral primitive, then either $(e_A \times e_B)\cdot e^i \leq 0$, $\forall e^i$ or $(e_A \times e_B)\cdot e^i \geq 0$, $\forall e^i$, $i \in \{0,1,2,3\}$.*

Proof. (By contradiction.) Suppose $(e_A \times e_B)\cdot e^0 > 0$ and $(e_A \times e_B)\cdot e^3 < 0$ (the other cases can be proved similarly). We consider the two prisms generated by $f^0 \oplus e_B$ and $f^3 \oplus e_A$ (Figure 5.3 (ii)). They share the quadrilateral $e_A \oplus e_B$ (shaded in the figure) and locate on different sides of it. Since both prisms are subsets of $A \oplus B$, $e_A \oplus e_B$ will not overlap with the boundary of $A \oplus B$. This contradicts the assumption that $e_A \oplus e_B$ is contributing. □

Propositions 5.1 and 5.2 give necessary conditions for contributing primitives by analyzing local supporting planes. If both input polyhedra are convex, these conditions become both necessary and sufficient (because local supporting planes of a convex polyhedron are also global supporting planes), and thus can be used to compute the complete Minkowski sum boundary. In the convex case they are equivalent to the criteria used to find boundary triangles and quadrilaterals in the slope-diagram based approaches [Ghosh, 1993, Fogel and Halperin, 2007, Barki et al., 2009a].

The above two propositions only check the relative positions of incident triangles. In this chapter we introduce two new propositions that check the orientation of incident triangles. These two propositions are based on the convexity of vertices and edges. For a vertex, if there exists a supporting plane that does not intersect the interior of the polyhedron in the infinitesimal neighborhood of the vertex, it is a convex vertex; otherwise, it is non-convex.

For an edge, if its dihedral angle is greater than π, it is a non-convex edge. These two new propositions cull out primitives that are generated by at least one non-convex vertex or edge. In [Barki et al., 2009a] the authors considered the case of non-convex edges, but they did not provide a proof.

Proposition 5.3. *Suppose $e_A \in E_A$ and $e_B \in E_B$. If either e_A or e_B is a non-convex edge, then $e_A \oplus e_B$ cannot be a contributing quadrilateral primitive.*

Proof. (By contradiction.) Suppose $e_A \oplus e_B$ is contributing, then $e_A \oplus e_B$ at least partially overlaps with the boundary of $A \oplus B$. Then there must exist a point $c \in e_A \oplus e_B$, such that c is on the boundary of $A \oplus B$ but not on any edge or vertex of the boundary (Figure 5.3 (iii)). Then c is either a local maximum or a local minimum of $A \oplus B$ in the direction of $e_A \times e_B$. Suppose $c = a + b$, $a \in e_A$ and $b \in e_B$, then both a and b should also be a local maximum or a local minimum of A and B respectively in the direction of $e_A \times e_B$. This cannot be true if either e_A or e_B is a non-convex edge. □

Proposition 5.4. *Suppose $f_A \in F_A$ and $v_B \in V_B$. If v_B is a non-convex vertex, then $f_A \oplus v_B$ cannot be a contributing triangle primitive.*

Proof. (By contradiction.) Suppose $f_A \oplus v_B$ is contributing, then $f_A \oplus v_B$ at least partially overlaps with the boundary of $A \oplus B$. Then there must exist a point $c \in f_A \oplus v_B$, such that c is on the boundary of $A \oplus B$ but not on any edge or vertex of the boundary (Figure 5.3 (iv)). Then c must be a local maximum of $A \oplus B$ in the direction of n_A. Suppose $c = a + v_B$, $a \in f_A$, then v_B must also be a local maximum of B in the direction of n_A. This cannot be true if v_B is a non-convex vertex. □

We use Proposition 5.1 and 5.4 to cull triangle primitives, and Proposition 5.2 and 5.3 to cull quadrilateral primitives. Note that not all the remaining primitives are contributing, because the four propositions only give necessary (not sufficient) conditions for contributing primitives. However, the number of primitives will be reduced greatly after culling. Table 5.1 compares the number of primitives before and after culling for several examples (see Figure 5.5 for pictures of these input models). We can see that less than 1% of the total primitives remain after culling. Figure 5.4 shows a Minkowski sum and its triangle and quadrilateral primitives after culling.

5.2.2 VBO Generation

To take advantage of back-face culling and still render the surface primitives correctly, we also need to compute their surface normals. From Proposition 5.1 and 5.2, we know that the actual normal of a triangle primitive $f_A \oplus v_B$ (equivalently $f_B \oplus v_A$) is always the same as the outward facing surface normal of f_A, and the actual normal of a quadrilateral primitive $e_A \oplus e_B$ is either $e_A \times e_B$ (when $(e_A \times e_B) \cdot e^i \leq 0, \forall e^i$) or $-e_A \times e_B$ (when $(e_A \times e_B) \cdot e^i \geq 0,$

A	B	#tri primitives			#quad primitives			#total
		before	after	%	before	after	%	%
bunny	ball	13 M	33 K	0.26%	29 M	18 K	0.06%	0.12%
pig	horse	114 M	692 K	0.61%	255 M	268 K	0.11%	0.26%
Scooby	torus	272 M	266 K	0.10%	612 M	327 K	0.05%	0.07%
dancing kids	octopus	651 M	1,755 K	0.27%	1,466 M	1,300 K	0.09%	0.14%

Table 5.1: Examples of surface primitive culling. From left to right, each column respectively shows models A and B, the number of triangle primitives before/after culling, the percentage of remaining triangle primitives, the number of quadrilateral primitives before/after culling, the percentage of remaining quadrilateral primitives, and the percentage of total remaining primitives after culling.

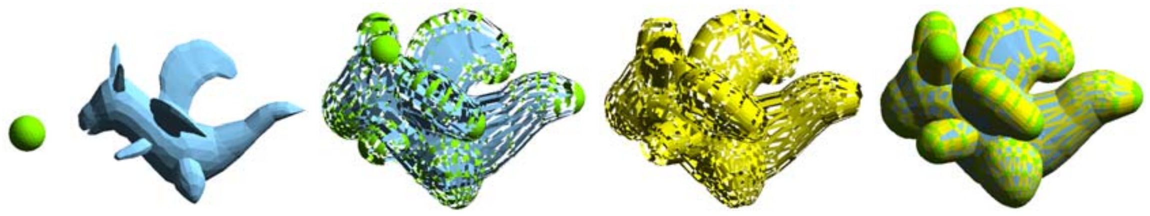


Figure 5.4: Triangle and quadrilateral primitives after culling. From left to right, each picture represents the two models (ball and dragon), triangle primitives after culling (two different colors represent triangles from the two different models), quadrilateral primitives after culling (yellow), and finally the rendered Minkowski sum.

$\forall e^i$). For implementation, we use a flag array to store the results of the culling test. If a surface primitive is noncontributing and should be culled out, we set its flag to be 0; otherwise, we set it to be 1 for triangle primitives, and 1 or -1 for quadrilateral primitives, according to whether its surface normal is $e_A \times e_B$ or $-e_A \times e_B$ respectively.

Since the number of surface primitives has quadratic complexity, the culling test will become costly in terms of running time when the sizes of the models increase. However, it can be easily parallelized since each surface primitive can be treated independently. We implemented the parallel culling test using NVIDIA's CUDA library on an NVIDIA Quadro 6000, which has 448 CUDA cores and a "compute capacity" of 2.0. We chose a block size of 16×16 based on the consideration that the maximum number of threads per block is 512. We also tested other block sizes (16×32 and 16×8) and found no performance improvement. Each block shares the data of 16 triangles and 16 vertices (or 16 edges from each of the two objects in the case of quadrilateral primitives), and performs culling tests on 256 surface primitives. For $0 \le i, j \le 15$, thread (i, j) works on triangle primitive $f_i \oplus v_j$ or quadrilateral primitive $e_i \oplus e_j$.

We have also implemented two memory optimization techniques: shared memory and coalesced global memory. Since all the 256 threads in a block share the coordinates of 16 triangles and 16 vertices (or 32 edges), and shared memory is much faster than global memory (if we do not consider memory caches), we copy these coordinates from global memory to the shared memory of each block before we perform the culling test. To achieve coalesced global memory access [CUDA, 2011], instead of storing the x, y, and z coordinate of each vertex consecutively $(x_1y_1z_1x_2y_2z_2...x_ny_nz_n)$, we store all the x coordinates first, followed by all y, and then all z coordinates $(x_1x_2...x_ny_1y_2...y_nz_1z_2...z_n)$. On a Quadro FX 5800, which does not support caches, we achieved a 3× speedup on average compared to the unoptimized version by using shared memory and coalesced global memory in our implementation. Out of the 3× speedup, 2× speedup comes from using shared memory and 1.5× speedup from using coalesced global memory. But on a Quadro 6000 with cache support, we achieved only a 1.13× speedup by using both shared memory and coalesced global memory, and almost 100% of the speedup comes from using coalesced global memory since the Quadro 6000 card itself includes memory cache support.

After all the culling tests are done, primitives with non-zero flags and their normals are written to the VBO, which is directly rendered using OpenGL. (A seemingly more efficient way to generate the VBO would be simply discarding noncontributing primitives and directly storing contributing ones to the VBO without using any flag array. However, this is difficult to implement on the GPU, because each primitive is tested independently in parallel and thus its position in the VBO cannot be determined at the time when it is tested.)

5.2.3 Rendering Results

We show some results of the above CUDA-based rendering algorithm in Figure 5.5. The program runs on 64-bit Windows 7 with an NVIDIA Quadro 6000 GPU with 6 GB video memory. We also implemented a sequential CPU version of the same algorithm and ran it on the same machine that has an Intel Core 2 Quad CPU at 2.66 GHz with 4 GB RAM, and compared the performance between the CUDA and the CPU implementation (see Table 5.2). Overall the CUDA implementation has a 40 to 50 times speedup over the CPU implementation.

We also applied the rendering algorithm to solid interpolation and shape morphing. The linear interpolation between two objects A and B can be computed using Minkowski sums as $(1-t)A \oplus tB$, $t \in [0, 1]$ (see [Kaul and Rossignac, 1992]). An example of shape morphing is shown in Figure 5.6. Here we use A to represent the cone model, B to represent the torus knot model, and render the Minkowski sums of $(1-t)A \oplus tB$ for t=0, 0.2, 0.4, 0.6, 0.8, and 1.

5.3 Voxelizing Minkowski Sums

In this section we introduce a new algorithm for voxelizing Minkowski sums, which is based on the rendering algorithm discussed in the previous section. Most existing GPU voxeliza-

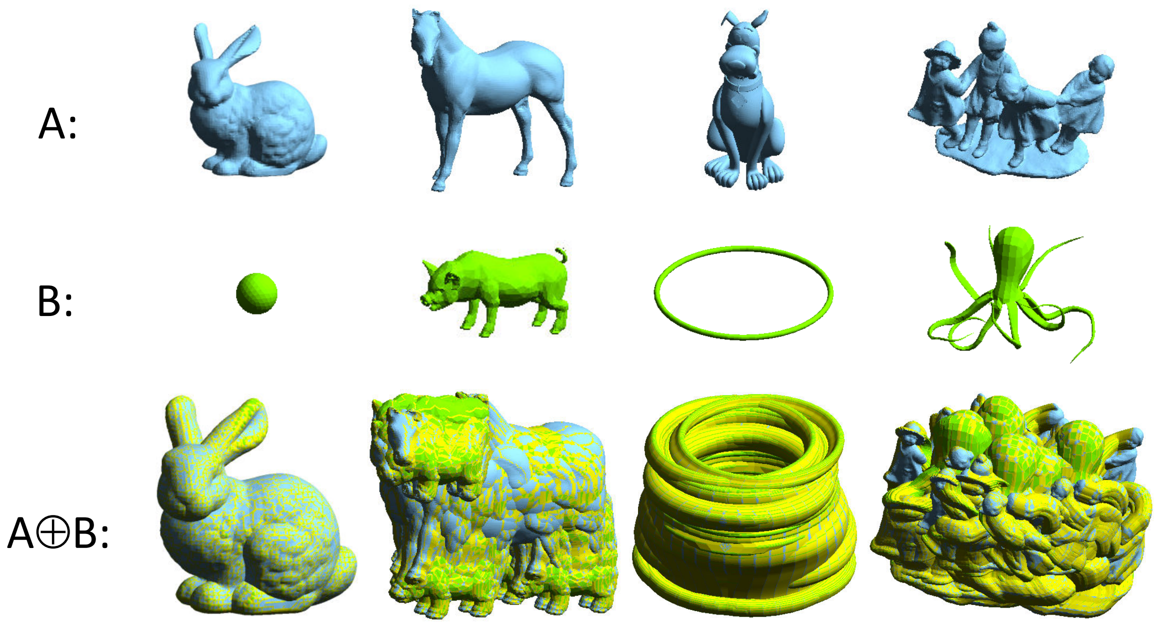


Figure 5.5: Four examples of CUDA-based Minkowski sum rendering. The Minkowski sum boundary is colored in green, blue, and yellow, representing triangle primitives from the green object, triangle primitives from the blue object, and quadrilateral primitives respectively.

tion algorithms utilize the GPU's rasterization functionality to voxelize boundary surfaces, and then perform a parity check via the stencil buffer [Llamas, 2007] or bitwise logic operations [Fang and Chen, 2000] to fill the interior voxels. However, they cannot be applied to the voxelization of Minkowski sums rendered using the above method, because of the existence of non-boundary surfaces in the interior. These surfaces will also be voxelized and cannot be distinguished from the actual boundary surfaces. The parity check will fail in such a case.

To solve this problem, instead of using parity checks to voxelize the interior, we propose using 3D flood fill to find all the *outer voxels* (defined as voxels whose centers are outside the Minkowski sum). The idea has similarities to the front propagation used for sweep volume approximation in [Kim et al., 2003]. Their method computes a discrete distance field of low resolution ($128 \times 128 \times 128$) on the GPU and then reads back the distance values to perform front propagation on the CPU. Our flood fill method directly uses the adjacency between neighboring voxels. It runs completely on the GPU and avoids the expensive readbacks from GPU to CPU memories.

Figure 5.7 gives a 2D illustration of our voxelization algorithm. It consists of three main steps: primitive voxelization, orthogonal fill, and flood fill, as discussed below.

A	B	#tri (A)	#tri (B)	CUDA (sec)	CPU (sec)	Speedup
bunny	ball	25,336	500	0.35	15.74	45×
pig	horse	2,784	40,746	2.71	113.49	42×
Scooby	torus	170,106	1,600	5.93	276.71	47×
dancing kids	octopus	78,706	8,276	13.31	563.57	42×

Table 5.2: Timing for rendering the Minkowski sums in Figure 5.5 (including primitive culling and VBO generation). From left to right, each column respectively shows models A and B, number of triangles of A and B, time of the CUDA implementation, time of the CPU implementation, and speedup of CUDA over CPU.

5.3.1 Primitive Voxelization

As the first step of the voxelization algorithm, we voxelize all the remaining surface primitives after culling. This gives an "incorrect" surface voxelization because, as discussed in section 5.2.1, we do not cull out all the noncontributing portions of surface primitives. There still exist primitives (or fractions of primitives) hidden inside by the boundary surfaces (see Figure 5.7 left), which are also voxelized along with the actual boundary primitives. However, this initial surface voxelization can be used later as a barrier to stop the flood fill. We will describe how to construct the final surface voxelization in section 5.3.3.

Graphics hardware is typically used for surface voxelization using the following technique. Each surface of an input model is projected orthogonally onto a 2D plane. The projected surfaces are rasterized to produce a set of fragments. These fragments contain depth information as well as 2D coordinates in the projection plane, which are used to map each fragment to a corresponding voxel in the 3D volume. In the slicing-based algorithm [Fang and Chen, 2000], each slice is voxelized consecutively by setting a near and far clipping plane. To generate a 1024^3 volume, the object has to be rendered 1024 times. This algorithm was improved by encoding each voxel into a single bit of a texel [Dong et al., 2004]. The benefits are twofold — it reduces both the memory cost and the number of passes needed to render the object.

We choose to use this voxelization technique for our surface voxelization due to its efficiency. Instead of using one or multiple 2D textures as in the original algorithm [Dong et al., 2004], we simplify the voxel access by using a single 3D texture, which has a 32 bit RGBA format and requires only 128MB video memory for the 1024^3 volumetric resolution. By using Multiple Render Targets (MRTs) we can render to 8 color buffers simultaneously. So in total we only need to render the VBO 4 (= 1024/32/8) times in order to voxelize all the surface primitives.

We set the view volume to be a little larger than the bounding box of the Minkowski sum, which can be easily computed by adding the bounding boxes (i.e., adding the corresponding minimum and maximum x, y, and z coordinates) of the two input models, such that all the voxels on the view volume boundary are outer voxels. To be more specific, we enlarge the

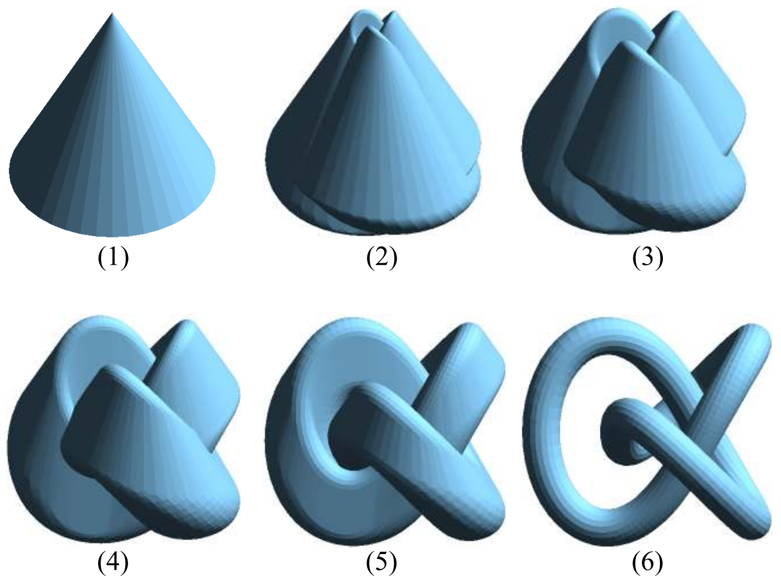


Figure 5.6: Shape morphing between a cone (78 triangles) and a torus knot (8000 triangles). The animation is computed and rendered at a framerate of 28.3 frames/second.

computed bounding box by a scale factor of $1 + 2(1 + \delta)/[N - 2(1 + \delta)]$, where N is the volumetric resolution and δ is a small number (0.1 in our implementation). This guarantees all the outer voxels are connected and can be visited from each other.

We implement the volume encoding through a fragment shader program, which computes an RGBA color for each fragment according to its depth information. The depth of each fragment is passed to the shader program as a texture coordinate, similar to the technique described by Fernando and Kilgard for Phong shading [Fernando and Kilgard, 2003].

A common problem of surface voxelization is that surfaces perpendicular (or nearly perpendicular) to the projection plane are not rasterized because their projections have a zero (or near zero) area. This problem was addressed in [Dong et al., 2004] by projecting the input model along three orthogonal directions and finally compositing the three directional voxelizations. We use the same approach and implement the composition using another fragment shader program. It samples the x and y 3D textures and writes to the z texture. The x and y textures are deleted after composition to free the video memory they use. In our experiments, the voxelization of Minkowski sum surface primitives (unculled triangles and quadrilaterals), including three directional projections and texture composition, takes on the order of one second (see Table 5.4). An example of primitive voxelization is shown in Figure 5.8.

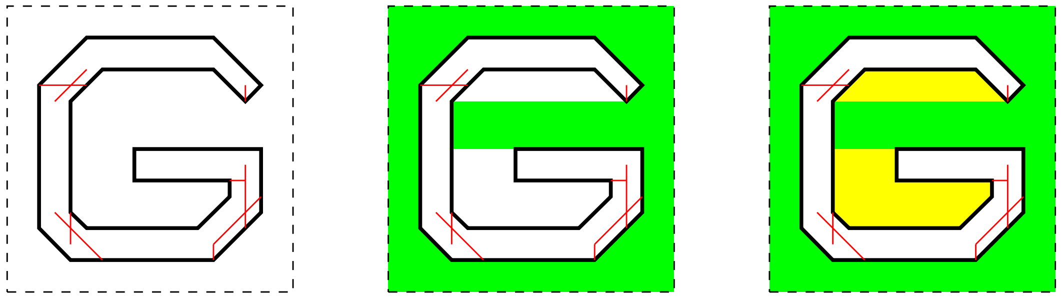


Figure 5.7: Overview of the voxelization algorithm. We first voxelize all the remaining surface primitives after culling (left), including boundary surfaces (solid black lines) and surfaces hidden inside (red lines). The outer dashed black lines represent the view volume. Then we perform an orthogonal fill along the six orthogonal directions (four in 2D) to find a portion of the set of outer voxels (in green, middle). Finally we use flood fill to find all the remaining outer voxels (in yellow, right).

5.3.2 Orthogonal Fill

The goal of orthogonal fill is to find all the outer voxels that are visible from the outside along the six orthogonal directions ($+x$, $+y$, $+z$, $-x$, $-y$, and $-z$). An example of such voxels is shown in green in Figure 5.9. The orthogonal fill is done by rendering the VBO (whose generation was described in section 5.2.2) six times, each time along a different orthogonal direction. These outer voxels serve as seeds for the later flood fill described in section 5.3.3.

The details of the orthogonal fill algorithm are as follows. We first generate a depth texture and attach it to a framebuffer object for offscreen rendering. Suppose we are rendering the VBO along the $axis$ direction ($axis$ is one of $+x$, $+y$, $+z$, $-x$, $-y$, and $-z$). We first need to rotate the unculled primitives (the VBO) such that $axis$ is aligned with the original $+z$ direction. Then we clear the depth buffer to the maximum depth value 1.0 and set the depth test to GL_LESS. Now we render the VBO to the depth texture. After rendering, the depth texture contains the smallest depth along the $axis$ direction sampled at the center of each pixel (Figure 5.9). Then we identify all the voxels with a depth (at their centers) no larger than the corresponding stored value in the depth texture as outer voxels, and write an appropriate RGBA color to a 3D texture for each pixel. For example, for the pixel with a smallest depth of 2.7 in Figure 5.9, the RGBA color is 11 10 00 00 in binary form. Here we use 1 for outer voxels and 0 otherwise. We only need three such 3D textures for the orthogonal fill – each pair of opposite directions share the same texture. After all six directions are computed, we composite the three directional 3D textures, using the same composition shader program that was used for primitive voxelization (section 5.3.1).

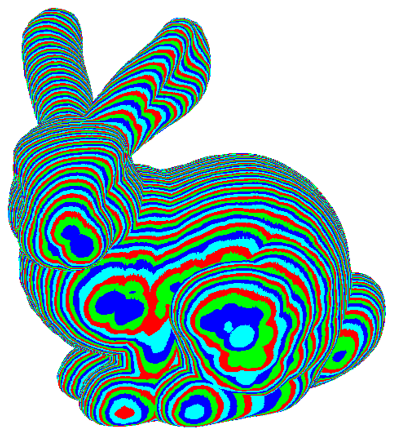


Figure 5.8: Primitive voxelization (1024^3) of the bunny $\oplus$ ball example in Figure 5.5. Each voxel is drawn as a point at its center. It is colored in red, green, blue, or cyan if its corresponding bit in the composite 3D texture is in the R, G, B, or A channel respectively. Note that each color band in the figure represents eight slices of voxels.

5.3.3 Flood Fill

After primitive voxelization and orthogonal fill, we have two 3D textures, one for primitive voxels and the other for the outer voxels found by orthogonal fill. Since these two steps only require three and six passes of rendering respectively and one pass of composition, they run very fast, both steps taking approximately one second for a resolution of 1024^3 (Table 5.4). However, they are "incomplete" voxelizations in that the primitive voxelization includes extra voxels hidden by the boundary surfaces, and the orthogonal fill voxelization contains only a portion of all the outer voxels. Outer voxels that are not visible from outside along any of the six orthogonal directions are not identified by the orthogonal fill process (for example, yellow voxels in Figure 5.7). To find all such outer voxels, we perform a flood fill that builds upon these two incomplete voxelizations.

In the following, we denote the 3D texture from primitive voxelization as T_b and the 3D texture from orthogonal fill as T_o. We use $T(i, j, k)$ to denote the bit in the 3D texture T that represents voxel (i, j, k). We call a voxel the *neighbor* of another if they share a face (according to this definition, a voxel has at most six neighbors).

Flood fill, also called seed fill, is one of the fundamental algorithms in raster graphics. Given a seed pixel inside a closed boundary, it recursively traverses all the pixels connected with it and assigns the desired color to them. Most flood fill algorithms explicitly or implicitly make use of a queue or stack data structure, both of which are difficult to implement efficiently on GPUs. What is more, our flood fill is performed in 3D image space, which increases the computational complexity. Flood fill algorithms are often sped up by filling whole lines instead of individual pixels [Burtsev and Kuzmin, 1993]. However, this technique relies on being able to perform valid parity checks, which we cannot support because of inte-

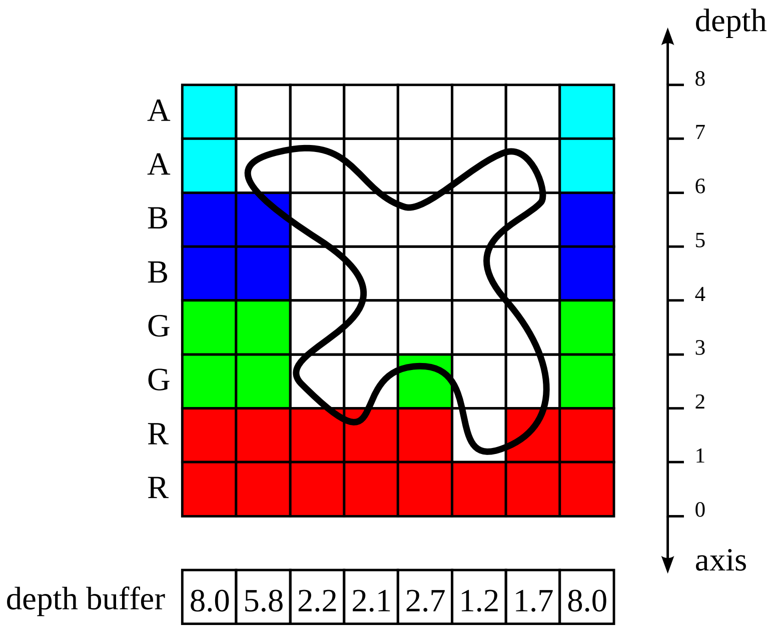


Figure 5.9: Outer voxels found by orthogonal fill along the specified axis direction. These outer voxels are colored in red, green, blue, or cyan according to their corresponding color channels. Here the volumetric resolution is 8^2 and each color channel has a bit-depth of 2. Note that the depth values, actually from $[0, 1]$, are scaled to $[0, 8]$ for the sake of clarity in the figure.

rior surface primitives. In this section, we propose a GPU-based 3D flood fill algorithm. It benefits from the three facts below. First, we use all the outer voxels from orthogonal fill as seeds, which usually have already covered a large portion of outer voxels. Second, we create a "mask" from newly found outer voxels such that we do not need to check every voxel in the next iteration. Third, the algorithm runs in parallel on the GPU, so in one iteration we are able to find a batch of new outer voxels, which represents a new "front."

Our flood fill is based on the following two observations: all the outer voxels are connected, and any neighbor of an outer voxel is either an outer voxel or a boundary voxel. Figure 5.10 shows a 2D illustration of the flood fill process. Outer voxels from orthogonal fill (in green) are used as "seeds" of the flood fill. Each iteration we find new outer voxels (in yellow) by checking the neighbors of existing outer voxels. The process is repeated until we reach the "barrier," the boundary voxels (in red), and no more new outer voxels are found.

Now we explain the implementation of flood fill in detail. Some voxels are marked as 1 in both T_o and T_b. As a preprocess, we need to reset them to 0 in T_o, since otherwise the flood fill will incorrectly penetrate into the interior of the object. This preprocess can be easily performed by adding T_b to T_o with logical operation GL_AND_INVERTED. Then we create two temporary 3D textures, T_{new} and T_{mask}, to store the outer voxels newly found

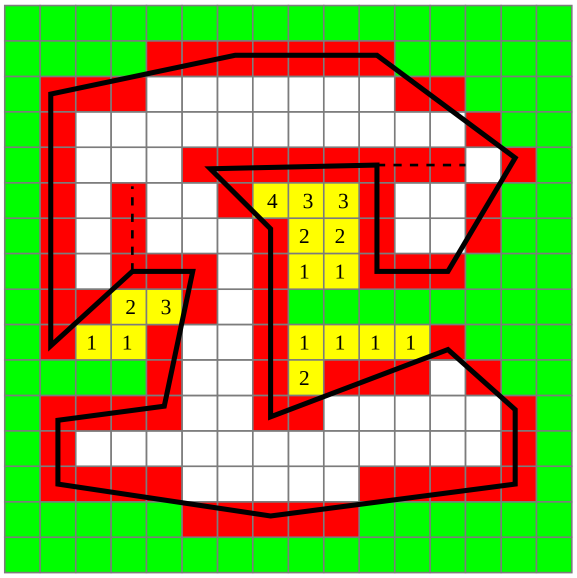


Figure 5.10: Flood fill of outer voxels. Voxels from primitive voxelization (T_b) are shown in red (we use solid black lines to represent the actual outer boundary of the Minkowski sum and dashed black lines the primitives hidden inside). Outer voxels found by orthogonal fill and not in T_b are shown in green. Yellow denotes outer voxels found by flood fill. The number in each yellow voxel indicates how many iterations are needed to reach that voxel.

in the current iteration and the voxel mask for the next iteration. For the first iteration, we check the neighbors of all the outer voxels in T_o. If they are neither already identified outer voxels in T_o nor boundary voxels in T_b, we add them to both T_{new} and T_o. Then we add all the neighbors of voxels in T_{new} to T_{mask}. We only need to check voxels in T_{mask} in the next iteration. Usually after the first iteration, the number of voxels we need to check will be greatly reduced. After each iteration, we employ an occlusion query to count the number of newly found voxels. If no new voxels are found, the flood fill is terminated and now T_o contains all the outer voxels. The pseudocode for our flood fill algorithm is given in Algorithm 5.1. The entire algorithm is implemented using three fragment shaders, for excluding voxels in T_b from T_o, checking neighbor voxels, and creating the mask respectively.

After all the outer voxels are identified, it becomes very easy to compute a correct surface voxelization. We only need to find those primitive voxels in T_b adjacent to an outer voxel. For example, in Figure 5.10, the final outer boundary surface (solid black lines) consists of primitive voxels (red) that have at least one outer voxel (green or yellow) as a neighbor.

The performance of flood fill is determined by the volumetric resolution and object complexity. For a resolution of $512 \times 512 \times 512$, it usually takes less than one second for most of the models we have tested (see Table 5.4). For a resolution of 1024^3, the time ranges from a few to tens of seconds, depending on how many iterations we need to perform the flood fill.

Algorithm 5.1 FloodFill

```
  **input:** $T_o, T_b$
  **output:** $T_o$
  $T_o \leftarrow T_o - (T_o \cap T_b)$
  create two 3D textures $T_{new}$ and $T_{mask}$;
  clear all voxels of $T_{mask}$ to 0;
  **for all** voxel(i,j,k) **do**
    **if** at least one neighbor has value 1 in $T_o$ **then**
      $T_{mask}(i,j,k) = 1$
    **end if**
  **end for**
  **repeat**
    clear all voxels of $T_{new}$ to 0
    **for all** voxel(i,j,k) satisfying $T_{mask}(i,j,k) = 1$ **do**
      **if** $T_o(i,j,k) = 0$ **and** $T_b(i,j,k) = 0$ **then**
        $T_o(i,j,k) = 1$
        $T_{new}(i,j,k) = 1$
      **end if**
    **end for**
    clear all voxels of $T_{mask}$ to 0
    **for all** voxel(i,j,k) **do**
      **if** at least one neighbor has value 1 in $T_{new}$ **then**
        $T_{mask}(i,j,k) = 1$
      **end if**
    **end for**
  **until** $\forall(i,j,k), T_{new}(i,j,k) = 0$ //test with occlusion query
  delete $T_{new}$ and $T_{mask}$
  **return** $T_o$
```

5.4 Robust Culling in the Presence of Floating Point Errors

The flood fill algorithm requires the computed Minkowski sum outer boundary to be "watertight," i.e., there can be no cracks in the outer boundary. Otherwise, the flood fill will wrongly penetrate into the interior of the Minkowski sum and make the voxelization algorithm fail. Theoretically, the Minkowski sum will be watertight as long as the two input models are watertight. However, cracks may occur due to floating point errors when performing surface primitive culling tests, as explained below.

The core of the four propositions used for surface primitive culling (section 5.2.1) relies on a 3D orientation test. For four points a, b, c, and d in $\mathbb{R}^3$, $\textsc{Orient3D}(a, b, c, d)$ returns a positive value if a, b, and c appear in clockwise order when viewed from d. To compute $\textsc{Orient3D}(a, b, c, d)$, we need to evaluate the sign of a 3×3 matrix determinant [Shewchuk, 1997], as shown below.

$$\textsc{Orient3D}(a, b, c, d) = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \tag{5.1}$$

The 3D orientation test may fail because of floating point rounding errors. To be more specific, if the above determinant is very close to zero (i.e., the four points are nearly coplanar), the computed sign may be incorrect and the orientation test will return a false answer. In such a case, some contributing surface primitives will be wrongly culled out. For example, when we perform the culling test for a triangle primitive, if at least one of its orientation tests returns a negative sign, we cull this triangle primitive out; thus it is possible that we cull out a contributing triangle primitive if one of its orientation tests wrongly returns a negative sign due to rounding errors. This will in turn cause cracks on the computed Minkowski sum boundary and the flood fill will penetrate into the interior of the Minkowski sum. A particularly challenging example to illustrate the rounding errors is to compute the Minkowski sum of a model and itself, where many orientation tests should return exact 0s, but actually return very small positive or negative values instead. Figure 5.11 (left) shows the Minkowski sum of two identical tessellated spheres. As we can see, quite a few boundary primitives are missing due to rounding errors.

Exact arithmetic is one solution to the floating point error problem. However, it comes at great performance expense, and implementing exact arithmetic on GPUs is not trivial. In this section, we compute an upper bound of the rounding error for equation (5.1). If the absolute value of the computed determinant is greater than the upper bound, we can safely return its sign as the result of the orientation test; otherwise, we are not sure whether the computed sign is correct or not, so we just keep this primitive without culling it. Compared to using exact arithmetic, we do not continue to compute a more accurate result when the determinant is within the error bound, which requires much more complex computations.

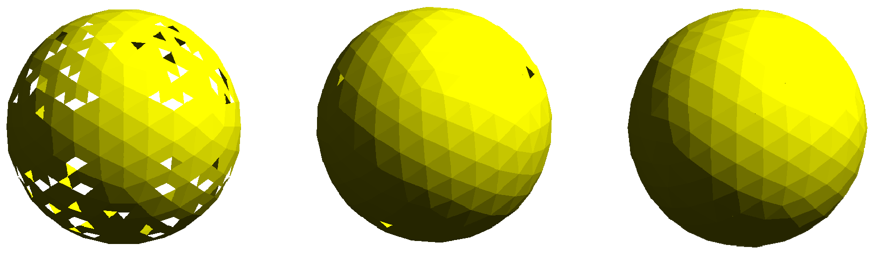


Figure 5.11: Computed Minkowski sum of two identical tessellated spheres, without rounding error analysis (left), by using Shewchuk's upper bound for the orientation test error (center), and by using our upper bound for the orientation test error (right).

An upper bound of the rounding error for equation (5.1) was introduced in [Shewchuk, 1997]. However we cannot directly apply it to our problem, because it is based on the assumption that all the floating point numbers in the matrix (5.1) are free of rounding errors. In our case, each of these numbers is the sum of two floating point quantities, one from each of the two input models, so they are already contaminated by rounding errors. Figure 5.11 (center) shows that some primitives are still missing if we directly apply the upper bound in [Shewchuk, 1997]. To derive the upper bound for our problem, we need to consider every operation that can introduce rounding errors, which in this case means replacing each element in matrix (5.1) with the sum of two original input numbers, as in equation (5.2), where numbers with subscript 1 or 2 represent coordinates from the first or second input model respectively.

Orient3D(a,b,c,d)

$$= \begin{vmatrix} (a_{x1}+a_{x2})-(d_{x1}+d_{x2}) & (a_{y1}+a_{y2})-(d_{y1}+d_{y2}) & (a_{z1}+a_{z2})-(d_{z1}+d_{z2}) \\ (b_{x1}+b_{x2})-(d_{x1}+d_{x2}) & (b_{y1}+b_{y2})-(d_{y1}+d_{y2}) & (b_{z1}+b_{z2})-(d_{z1}+d_{z2}) \\ (c_{x1}+c_{x2})-(d_{x1}+d_{x2}) & (c_{y1}+c_{y2})-(d_{y1}+d_{y2}) & (c_{z1}+c_{z2})-(d_{z1}+d_{z2}) \end{vmatrix} \tag{5.2}$$

$$= \begin{vmatrix} (a_{x1}-d_{x1})+(a_{x2}-d_{x2}) & (a_{y1}-d_{y1})+(a_{y2}-d_{y2}) & (a_{z1}-d_{z1})+(a_{z2}-d_{z2}) \\ (b_{x1}-d_{x1})+(b_{x2}-d_{x2}) & (b_{y1}-d_{y1})+(b_{y2}-d_{y2}) & (b_{z1}-d_{z1})+(b_{z2}-d_{z2}) \\ (c_{x1}-d_{x1})+(c_{x2}-d_{x2}) & (c_{y1}-d_{y1})+(c_{y2}-d_{y2}) & (c_{z1}-d_{z1})+(c_{z2}-d_{z2}) \end{vmatrix} \tag{5.3}$$

Further analysis indicates that equation (5.2) suffers from so-called "subtractive cancellation," which happens when two nearly equal numbers contaminated by rounding errors are subtracted. This causes relative errors already present in these two numbers to be magnified. In equation (5.2), a_{x1} and d_{x1} are x-coordinates of adjacent vertices from the first model. For densely tessellated models, usually $a_{x1} \approx d_{x1}$, and similarly $a_{x2} \approx d_{x2}$. Thus $a_{x1} + a_{x2} \approx d_{x1} + d_{x2}$. Subtractive cancellation therefore occurs when we compute $(a_{x1} + a_{x2}) - (d_{x1} + d_{x2})$. The same is true for all the nine elements of the matrix in (5.2). To avoid this undesirable subtractive cancellation, we rewrite equation (5.2) as (5.3). Since

a_{x1} and d_{x1} are free of rounding errors, there is no subtractive cancellation in $a_{x1} - d_{x1}$. Furthermore, the revised formulation benefits from the fact that if two inputs p and q are rounding-error free floating point numbers and sufficiently close (to be more specific, $q/2 \leq p \leq 2q$), the subtraction $p - q$ is exact.

Rounding error analysis starts with computing ϵ, a quantity called "unit roundoff," which is half the distance between 1 and the next larger representable floating point number. For IEEE 754 single precision arithmetic, $\epsilon = 2^{-24}$; for double precision, $\epsilon = 2^{-53}$. Under IEEE floating point arithmetic, the relative rounding error of all basic arithmetic operations (+, –, *, /) cannot exceed the unit roundoff. More formally, if we use $fl(\cdot)$ to denote the evaluation of an expression $(\cdot)$ in floating point arithmetic, then for two rounding-error free floating point numbers x and y, the arithmetic operations op (+, –, *, /) satisfy

$$fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\delta| \leq \epsilon. \tag{5.4}$$

The above equation is the basic model for most rounding error analysis.

We take a simple example, $x^2 - y^2$, to explain how to perform rounding error analysis using model (5.4). Note that there are three floating point operations in the expression $x^2 - y^2$: square of x, square of y, and subtraction. Applying model (5.4) to each of the three operations, we have

$$\begin{aligned}
&fl(x^2 - y^2)\\
&= \left(x^2\left(1+\delta_1\right) - y^2\left(1+\delta_2\right)\right)\left(1+\delta_3\right)\\
&= x^2\left(1+\delta_1+\delta_3+\delta_1\delta_3\right) - y^2\left(1+\delta_2+\delta_3+\delta_2\delta_3\right),
\end{aligned}$$

where $|\delta_i| \leq \epsilon$, for $i = 1$, 2, and 3. Then the rounding error of $x^2 - y^2$ is

$$\begin{aligned}
&|fl(x^2 - y^2) - (x^2 - y^2)|\\
&= \left|x^2\left(\delta_1+\delta_3+\delta_1\delta_3\right) - y^2\left(\delta_2+\delta_3+\delta_2\delta_3\right)\right|\\
&\leq x^2\left|\delta_1+\delta_3+\delta_1\delta_3\right| + y^2\left|\delta_2+\delta_3+\delta_2\delta_3\right|\\
&\leq (2\epsilon+\epsilon^2)(x^2+y^2).
\end{aligned} \tag{5.5}$$

Unfortunately, to carry out the rounding error analysis for equation (5.3), it would be tedious and error-prone to compute the rounding error step by step using model (5.4), as above, since equation (5.3) requires 41 floating point operations, 18 more operations than were analyzed in [Shewchuk, 1997]. To simplify the process, we use a convenient notation described in [Higham, 2002, chapter 3]:

$$\langle k\rangle = \prod_{i=1}^{k}(1+\delta_i), \quad |\delta_i| \leq \epsilon. \tag{5.6}$$

Here $\langle k\rangle$ serves as a relative error counter. Note $\langle j\rangle\langle k\rangle = \langle j+k\rangle$.

Again, we take $x^2 - y^2$ as an example. We have

$$\begin{aligned} fl(x^2 - y^2) &= (x^2 \langle 1 \rangle - y^2 \langle 1 \rangle) \langle 1 \rangle \\ &= x^2 \langle 2 \rangle - y^2 \langle 2 \rangle . \end{aligned} \tag{5.7}$$

The three $\langle 1 \rangle$s in the first step correspond to the three floating point operations: square of x, square of y, and subtraction. So the rounding error of $x^2 - y^2$ is bounded by

$$\begin{aligned} \left| fl(x^2 - y^2) - (x^2 - y^2) \right| &= \left| x^2 (\langle 2 \rangle - 1) - y^2 (\langle 2 \rangle - 1) \right| \\ &\leq |\langle 2 \rangle - 1| (x^2 + y^2). \end{aligned}$$

Note that the two $\langle 2 \rangle$s in the first step represent different numbers, i.e., they have different δ_is in equation (5.6). This explains why it is $x^2 + y^2$ instead of $x^2 - y^2$ in the second step.

As we can see from above, at the end of rounding error analysis, it is necessary to bound $|\langle k \rangle - 1|$. A useful inequality is proved in [Higham, 2002, chapter 3]:

$$|\langle k \rangle - 1| \leq 1.01 k\epsilon, \quad \text{if } k\epsilon \leq 0.01. \tag{5.8}$$

The condition $k\epsilon \leq 0.01$ is always true unless k is enormous, since ϵ is very small for IEEE floating point arithmetic. Using this inequality we can compute an upper bound of the rounding error of $x^2 - y^2$ as below:

$$\begin{aligned} |fl(x^2 - y^2) - (x^2 - y^2)| &\leq |\langle 2 \rangle - 1| (x^2 + y^2) \\ &\leq 2.02\epsilon(x^2 + y^2). \end{aligned} \tag{5.9}$$

Compared to the upper bound in (5.5), the above upper bound derived using the $\langle k \rangle$ notation is a little larger, but it greatly simplifies the derivation process, especially when there are many floating point operations, as in our case.

Following a similar process, we can derive an upper bound for the rounding error of equation (5.3), as shown in the proposition below.

Proposition 5.5. *The rounding error of equation (5.3) is bounded by*

$$\begin{aligned} |\mathrm{err}| \leq \; & 11.11\epsilon \big[\overline{\mathrm{ad}}_z \cdot (\overline{\mathrm{bd}}_x \cdot \overline{\mathrm{cd}}_y + \overline{\mathrm{cd}}_x \cdot \overline{\mathrm{bd}}_y) \\ & + \overline{\mathrm{bd}}_z \cdot (\overline{\mathrm{cd}}_x \cdot \overline{\mathrm{ad}}_y + \overline{\mathrm{ad}}_x \cdot \overline{\mathrm{cd}}_y) \\ & + \overline{\mathrm{cd}}_z \cdot (\overline{\mathrm{ad}}_x \cdot \overline{\mathrm{bd}}_y + \overline{\mathrm{bd}}_x \cdot \overline{\mathrm{ad}}_y) \big] \end{aligned} \tag{5.10}$$

where $\overline{\mathrm{ad}}_z = |a_{z1} - d_{z1}| + |a_{z2} - d_{z2}|$, *etc.*

Proof. Let the right side of equation (5.3) be P. If we define

$$\begin{aligned} ad_{z1} &= a_{z1} - d_{z1}, \\ ad_{z2} &= a_{z2} - d_{z2}, \\ ad_z &= ad_{z1} + ad_{z2}, \end{aligned} \tag{5.11}$$

etc., then P can be rewritten as

$$P = \begin{vmatrix} ad_x & ad_y & ad_z \\ bd_x & bd_y & bd_z \\ cd_x & cd_y & cd_z \end{vmatrix}. \tag{5.12}$$

From model (5.4) and notation (5.6), we have

$$\begin{aligned} fl(ad_{z1}) &= (a_{z1} - d_{z1}) \langle 1 \rangle = ad_{z1} \langle 1 \rangle \\ fl(ad_{z2}) &= (a_{z2} - d_{z2}) \langle 1 \rangle = ad_{z2} \langle 1 \rangle \\ fl(ad_z) &= \big(fl(ad_{z1}) + fl(ad_{z2})\big) \langle 1 \rangle \\ &= (ad_{z1} \langle 1 \rangle + ad_{z2} \langle 1 \rangle) \langle 1 \rangle \\ &= ad_{z1} \langle 2 \rangle + ad_{z2} \langle 2 \rangle . \end{aligned} \tag{5.13}$$

Thus the rounding error of ad_z is

$$\begin{aligned} |err(ad_z)| &= |fl(ad_z) - ad_z| \\ &= |ad_{z1} \langle 2 \rangle + ad_{z2} \langle 2 \rangle - ad_{z1} - ad_{z2}| \\ &= |ad_{z1}(\langle 2 \rangle - 1) + ad_{z2}(\langle 2 \rangle - 1)| \\ &\leq |\langle 2 \rangle - 1| \cdot (|ad_{z1}| + |ad_{z2}|) . \end{aligned} \tag{5.14}$$

If we define

$$\overline{ad}_z = |ad_{z1}| + |ad_{z2}| , \tag{5.15}$$

equation (5.14) can be rewritten as

$$|err(ad_z)| \leq |\langle 2 \rangle - 1| \cdot \overline{ad}_z. \tag{5.16}$$

Note that the two $\langle 2 \rangle$s in equation (5.13) represent different numbers, i.e., they have different δ_is in notation (5.6). For convenience of notation, we do not distinguish the difference between them and rewrite equation (5.13) as

$$\begin{aligned} fl(ad_z) &= (ad_{z1} + ad_{z2}) \langle 2 \rangle \\ &= ad_z \langle 2 \rangle . \end{aligned} \tag{5.17}$$

As we can see, by using this change of notation, we get a very concise expression $fl(ad_z) = ad_z \langle 2 \rangle$. However, when we recover the magnitude of the rounding error $|fl(ad_z) - ad_z|$ from equation (5.17), we must use $\overline{ad}_z$, as shown in equation (5.16), instead of $|ad_z|$. We will use the same convention in all the following derivations. Similarly, equations (5.16) and (5.17) hold for all the elements in matrix (5.12).

Now we define the following variables that represent intermediate steps in calculating P:

$$\begin{aligned} M_1 &= bd_x \cdot cd_y - cd_x \cdot bd_y && (5.18)\\ M_2 &= cd_x \cdot ad_y - ad_x \cdot cd_y && (5.19)\\ M_3 &= ad_x \cdot bd_y - bd_x \cdot ad_y && (5.20)\\ N_1 &= ad_z \cdot M_1 && (5.21)\\ N_2 &= bd_z \cdot M_2 && (5.22)\\ N_3 &= cd_z \cdot M_3. && (5.23) \end{aligned}$$

Then we have

$$P = N_1 + N_2 + N_3. \tag{5.24}$$

Consider the rounding error of M_1. We have

$$\begin{aligned} & fl(M_1) \\ & = (fl(bd_x) \cdot fl(cd_y) \langle 1 \rangle - fl(cd_x) \cdot fl(bd_y) \langle 1 \rangle) \langle 1 \rangle \\ & = (bd_x \langle 2 \rangle \cdot cd_y \langle 2 \rangle \langle 1 \rangle - cd_x \langle 2 \rangle \cdot bd_y \langle 2 \rangle \langle 1 \rangle) \langle 1 \rangle \\ & = (bd_x \cdot cd_y - cd_x \cdot bd_y) \langle 6 \rangle \\ & = M_1 \langle 6 \rangle . \end{aligned} \tag{5.25}$$

The magnitude of the rounding error of M_1 is

$$|err(M_1)| = |fl(M_1) - M_1| \leq |\langle 6 \rangle - 1| \cdot \overline{M}_1, \tag{5.26}$$

where

$$\overline{M}_1 = \overline{bd}_x \cdot \overline{cd}_y + \overline{cd}_x \cdot \overline{bd}_y. \tag{5.27}$$

Here $\overline{bd}_x$ etc. are defined in the same way as $\overline{ad}_z$ in equation (5.15). Note that the minus sign from equation (5.18) becomes plus in the above equation since we are taking absolute values. Again we should use $\overline{M}_1$ instead of $|M_1|$ for the same reason as we use $\overline{ad}_z$ in equation (5.16).

Now consider the rounding error of N_1. We have

$$\begin{aligned} fl(N_1) &= fl(ad_z) \cdot fl(M_1) \langle 1 \rangle \\ &= ad_z \langle 2 \rangle \cdot M_1 \langle 6 \rangle \langle 1 \rangle \\ &= N_1 \langle 9 \rangle . \end{aligned} \tag{5.28}$$

If we define

$$\overline{N}_1 = \overline{ad}_z \cdot \overline{M}_1, \tag{5.29}$$

the rounding error of N_1 is

$$|err(N_1)| \leq |\langle 9 \rangle - 1| \cdot \overline{N}_1. \tag{5.30}$$

Analogous results hold for N_2 and N_3.

Finally we consider the rounding error of P in equation (5.24). Note that its rounding error depends on the order in which the two sums are performed. Here we assume that it is computed from left to right, i.e., $P = (N_1 + N_2) + N_3$. Later we will loosen the error bound to eliminate the dependence on the operation order. We have

$$\begin{aligned} fl(P) &= \Big(\big(fl(N_1) + fl(N_2) \big) \langle 1 \rangle + fl(N_3) \Big) \langle 1 \rangle \\ &= \big((N_1 \langle 9 \rangle + N_2 \langle 9 \rangle) \langle 1 \rangle + N_3 \langle 9 \rangle \big) \langle 1 \rangle \\ &= N_1 \langle 11 \rangle + N_2 \langle 11 \rangle + N_3 \langle 10 \rangle . \end{aligned} \tag{5.31}$$

To make equation (5.31) symmetric for N_1, N_2 and N_3, we can add one extra $\langle 1 \rangle$ to N_3. This will slightly loosen the error bound, as shown below, but it makes the expression easier to compute and also independent of the operation order:

$$\begin{aligned} |err(P)| &= |fl(P) - P| \\ &= |N_1 \langle 11 \rangle + N_2 \langle 11 \rangle + N_3 \langle 10 \rangle - P| \\ &\leq |N_1 \langle 11 \rangle + N_2 \langle 11 \rangle + N_3 \langle 11 \rangle - P| \\ &= |P \langle 11 \rangle - P| . \end{aligned} \tag{5.32}$$

Now we define

$$\overline{P} = \overline{N}_1 + \overline{N}_2 + \overline{N}_3. \tag{5.33}$$

The rounding error of P is

$$|err(P)| \leq |\langle 11 \rangle - 1| \, \overline{P}. \tag{5.34}$$

Using inequality (5.8), we get

$$\begin{aligned} |err(P)| &\leq 11.11\epsilon\overline{P} \\ &= 11.11\epsilon \left(\overline{N}_1 + \overline{N}_2 + \overline{N}_3 \right) \\ &= 11.11\epsilon \left(\overline{ad}_z \cdot \overline{M}_1 + \overline{bd}_z \cdot \overline{M}_2 + \overline{cd}_z \cdot \overline{M}_3 \right) \\ &= 11.11\epsilon \big[\overline{ad}_z \cdot (\overline{bd}_x \cdot \overline{cd}_y + \overline{cd}_x \cdot \overline{bd}_y) \\ &\quad + \overline{bd}_z \cdot (\overline{cd}_x \cdot \overline{ad}_y + \overline{ad}_x \cdot \overline{cd}_y) \\ &\quad + \overline{cd}_z \cdot (\overline{ad}_x \cdot \overline{bd}_y + \overline{bd}_x \cdot \overline{ad}_y) \big] \end{aligned} \tag{5.35}$$

This proves the error bound given in equation (5.10). □

When we perform culling tests, we check the computed determinant of matrix (5.3) against its error bound computed using the above inequality (5.10). If the absolute value of the determinant is greater than the error bound, we return its sign as the result of the orientation test; otherwise, we just keep the corresponding primitive without culling it. Figure 5.11 (right) shows the result after applying the adaptive 3D orientation tests. It is easy to implement on GPUs, and does not cause any significant performance difference, since the rounding error check affects only those orientation tests where the four points are nearly coplanar, and flood fill dominates running times. For example, for the inputs in Table 5.4 and 5.5, times are at most 2% slower with robust culling (see Table 5.6 at the end of the chapter). The number of remaining primitives after culling increases very little ($\sim$1%) compared with using orientation tests without checking rounding error. Furthermore, we simply check the error bound instead of computing an exact result, which would be much more time consuming.

5.5 Results and Performance

Figure 5.12 shows the voxelization results of the four Minkowski sums in Figure 5.5. The timings under two different resolutions are given in Table 5.3 and 5.4. All the timings reported by us in this section are obtained by taking the average of five repeated experiments. Here again we use the same NVIDIA Quadro 6000 GPU and Intel Core 2 Quad CPU. The program runs on CUDA driver 3.2 and 64-bit Windows 7. We can see that for complex models with tens or hundreds of thousands of triangles, we can compute their Minkowski sums within one minute. The performance is mainly dominated by VBO generation and flood fill. The VBO generation time is nearly proportional to the sizes of the input models, since we need to test every surface primitive. The flood fill time is determined by the shape complexity of the Minkowski sum. To be more specific, if a large portion of its boundary surface is invisible along all the orthogonal directions from outside, the flood fill will take more time. This can be easily seen by comparing bunny $\oplus$ ball and Scooby $\oplus$ torus in Figure 5.5.

Below we compare our voxelization approach with three other recent approaches for approximate Minkowski sum computation, the distance-field based approach [Varadhan and Manocha, 2006], the point based approach [Lien, 2008a], and the FFT based approach [Lysenko et al., 2011]. The approach in [Varadhan and Manocha, 2006] outputs watertight boundary surfaces that are extracted as isosurfaces from the distance field. The point based approach [Lien, 2008a] generates discrete sample points covering the Minkowski sum boundary. Similar to our approach, the FFT based approach [Lysenko et al., 2011] generates voxelized Minkowski sums, but it first voxelizes the two input models, and then computes the Minkowski sum of the two voxelizations. Our approach directly creates surface and solid voxelizations of the Minkowski sum from the boundaries of the input models. The accuracy of the distance-field approach, the FFT based approach, and our approach is governed by the resolution of the volumetric grid. The resolutions used in the distance field approach range from 32^3 to 128^3, lower than the 512^3 and 1024^3 resolutions used in our approach (distance fields require storing distances for each cell vertex, whereas we only store one bit per cell). The FFT based approach can only achieve a resolution of 256^3 due to memory limitations (it stores a floating point number for each voxel). The accuracy of the point based approach is determined by the sampling density. The sampling densities reported in [Lien, 2008a] are

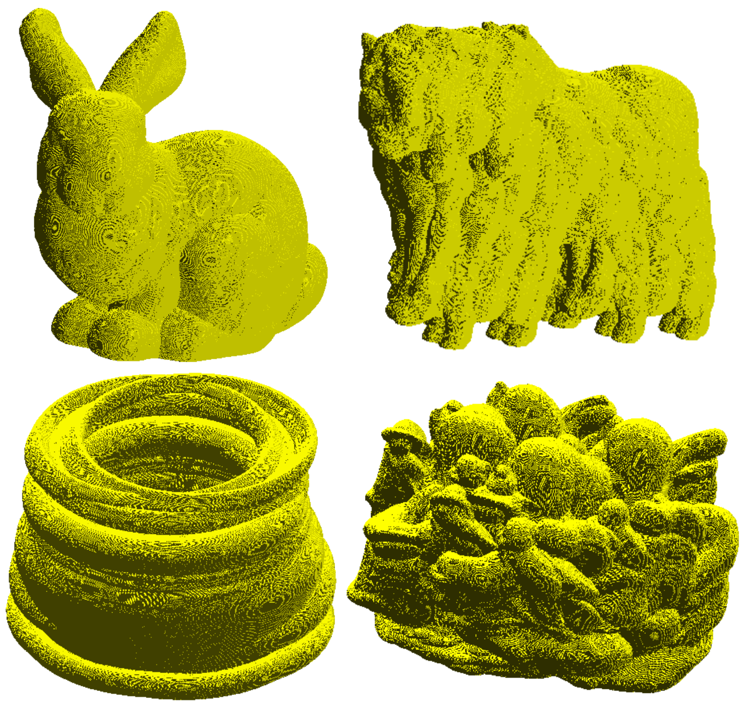


Figure 5.12: Voxelization (1024^3) of the four Minkowski sums in Figure 5.5.

equivalent to volumetric resolutions ranged from 64^3 to 256^3, also lower than ours (again their samples have more than one bit of information). The distance-field approach guarantees that their approximation has the same topology as the exact Minkowski sum, while the point based approach, the FFT based approach, and our approach do not provide such topological guarantees.

We next compare the performance of our voxelization approach with the method proposed by Lien [Lien, 2008d]. We use the same test models as in [Lien, 2008d] and report the test results in Table 5.5, with floating point error checking (note that Lien does not check floating point errors). The test models and their Minkowski sums are shown in Figure 5.13. For Lien's method, we use the timings reported in [Lien, 2008d] for comparison, which were obtained on a PC with two Intel Core 2 CPUs at 2.13 GHz and 4 GB memory. Since our algorithm runs completely on the GPU, its performance is mainly determined by the GPU instead of the CPU. Here we again use the Quadro 6000 GPU as detailed above for our timings. From Table 5.5, we can see that our approach is at least one order of magnitude faster. Lien's approach handles enclosed voids and generates exact boundary representations except that it does not produce low dimensional boundaries. Our voxelization approach is an approximate method. However, we can achieve relatively high accuracy by supporting a resolution of 1024^3. Most test models used here are generated by polygonizing models with curved surfaces. Even a simple curved object like a sphere would need to be polygonized with

$A \oplus B$	VBO	P.V.	O.F.	F.F.	#F.F.	Total
bunny (25,336) $\oplus$ ball (500)	0.35	0.10	0.12	0.31	0	0.89
pig (2,784) $\oplus$ horse (40,746)	2.76	1.02	1.02	0.58	23	5.37
Scooby (170,106) $\oplus$ torus (1,600)	5.98	0.69	0.68	0.90	44	8.25
dancing kids (78,706) $\oplus$ octopus (8,276)	13.47	3.38	3.31	0.95	41	21.10

Table 5.3: Timing for voxelizing the four Minkowski sums in Figure 5.5 under the resolution of 512^3 (in seconds). From left to right, each column respectively shows the input models with their numbers of triangles, time for VBO generation (including primitive culling), time for primitive voxelization, time for orthogonal fill, time for flood fill, number of flood fill iterations, and total time.

$A \oplus B$	VBO	P.V.	O.F.	F.F.	#F.F.	Total
bunny (25,336) $\oplus$ ball (500)	0.35	0.41	0.41	2.25	1	3.42
pig (2,784) $\oplus$ horse (40,746)	2.76	2.17	1.29	7.76	111	13.98
Scooby (170,106) $\oplus$ torus (1,600)	5.98	1.57	0.96	10.91	153	19.42
dancing kids (78,706) $\oplus$ octopus (8,276)	13.47	6.89	3.57	12.54	185	36.46

Table 5.4: Timing for voxelizing the four Minkowski sums in Figure 5.5 under the resolution of 1024^3 (in seconds). From left to right, each column respectively shows the input models with their numbers of triangles, time for VBO generation (including primitive culling), time for primitive voxelization, time for orthogonal fill, time for flood fill, number of flood fill iterations, and total time.

about 5,000 triangles [Baumgardner and Frederickson, 1985] in order to match the accuracy of the voxelization at a resolution of 1024^3.

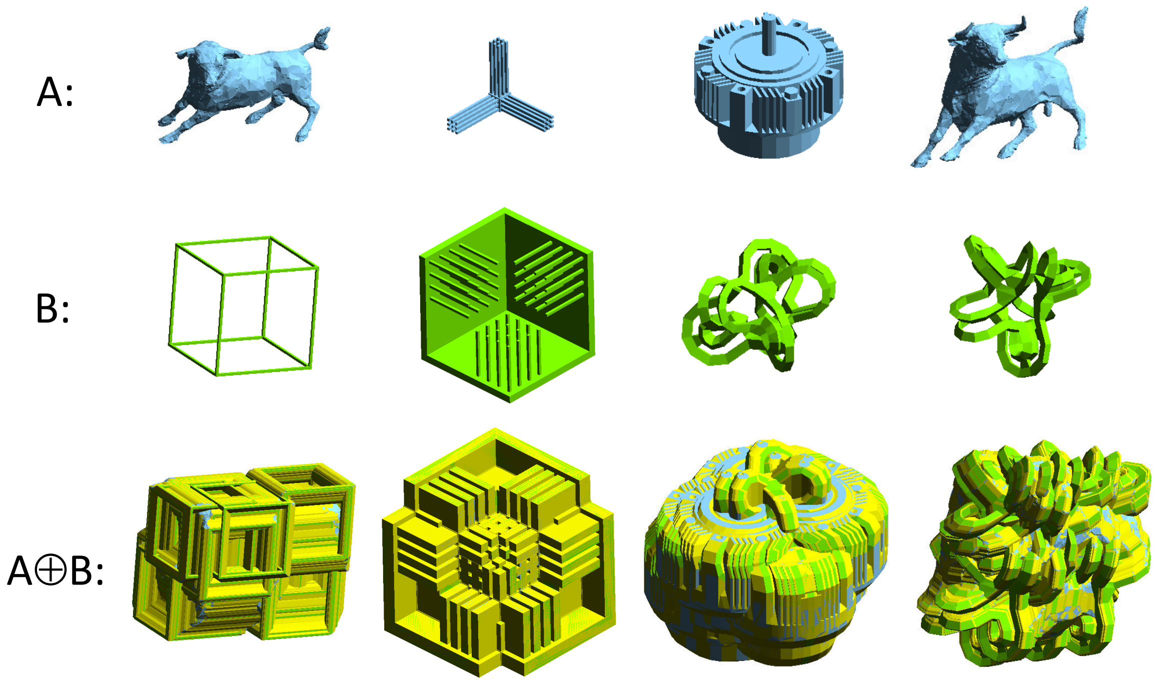


Figure 5.13: The test models used in Table 5.5 for performance comparison with Lien's approach.

We also found, from the source code Lien kindly provided to us for performance testing, that he also used Proposition 5.3 and 5.4 for primitive culling. However, they were not covered in his paper.

To measure the slowdown of the performance caused by the adaptive robust culling, we use a naïve implementation of equation (5.2) without checking floating point errors and compare the timings of computing the eight test cases in Table 5.4 and 5.5. The results are shown in Table 5.6, from which we can see that the slowdown caused by using robust culling is less than 2%. For the "grate1 $\oplus$ grate2" and "Scooby $\oplus$ torus" examples, if we do not use robust culling there will be cracks on the computed outer boundaries, therefore we use a small ϵ (0.001) as a threshold for the 3D orientation tests in the non-robust implementation to ensure correct outputs (to be more specific, we return a positive value only when the computed result of equation (5.2) is greater than ϵ, and a negative value only when it is less than $-\epsilon$). The "grate1 $\oplus$ grate2" test case is faster with robust culling because fewer primitives remain after adaptive culling tests and therefore the voxelization process takes less time.

A	B	#tri(A)	#tri(B)	Lien's	#Flood Fill		Ours		Speedup	
					512	1024	512	1024	512	1024
bull	frame	12,396	96	289.30	120	240	2.14	18.07	135×	16×
grate1	grate2	540	942	318.50	0	0	2.17	7.68	147×	41×
clutch	knot	2,116	992	347.00	0	0	0.94	4.31	370×	81×
bull	knot	12,396	992	755.10	113	195	2.98	15.68	254×	48×

Table 5.5: Performance comparison with Lien's approach (in seconds). From left to right, each column respectively shows model A and B, number of triangles of A and B, time of Lien's approach, number of flood fill iterations, time of our approach, and the speedup. The "512" and "1024" subcolumns represent 512^3 and 1024^3 resolutions.

A	B	512			1024		
		Nonrobust	Robust	%slower	Nonrobust	Robust	%slower
bull	frame	2.1105	2.1384	1.32%	17.9500	18.0656	0.64%
grate1	grate2	2.2334	2.1663	-3.00%	7.7638	7.6780	-1.10%
clutch	knot	0.9286	0.9379	1.00%	4.3050	4.3090	0.09%
bull	knot	2.9904	2.9764	-0.47%	15.5269	15.6843	1.01%
bunny	ball	0.8919	0.8875	-0.49%	3.4248	3.4181	-0.20%
horse	pig	5.2694	5.3725	1.96%	13.9405	13.9808	0.29%
Scooby	torus	8.1863	8.2519	0.80%	19.3996	19.4221	0.12%
dancing kids	octopus	20.8011	21.1049	1.46%	36.2551	36.4576	0.56%

Table 5.6: Performance comparison of computing voxelized Minkowski sums with robust and non-robust culling (in seconds). From left to right, each column respectively shows model A and B, timing with non-robust culling, timing with robust culling, and the slowdown due to using robust culling (negative values mean the algorithm is faster with robust culling). The "512" and "1024" columns represent 512^3 and 1024^3 resolutions.

5.6 Conclusions

We have presented a new approach for directly computing a voxelization of the Minkowski sum of two polyhedral objects, without having to compute a complete boundary representation. By analyzing and adaptively bounding the floating point rounding errors in computing the predicate we use for culling surface primitives, we guarantee that no primitives belonging to the actual Minkowski sum boundary will be mistakenly culled. Our voxelization approach avoids complex 3D Boolean operations by utilizing the GPU's rasterization functionality. The whole algorithm runs in parallel on the GPU and is at least one order of magnitude faster than existing algorithms at the relatively high resolution of 1024^3. It is memory efficient and able to handle large geometric models.

Chapter 6

Applications of Voxelized Minkowski Sums

In section 1.3 and 2.1, we discussed some applications of Minkowski sums including solid modeling, motion planning, and penetration depth computation. In this chapter, we give some examples of these applications computed using the two voxelization algorithms described in chapter 4 and 5. All the voxelized Minkowski sums are computed on an Intel Core 2 Quad CPU at 2.66 GHz with 4 GB RAM and a Quadro 6000 GPU with 6 GB video memory. The program runs on 64-bit Windows 7.

6.1 Solid Modeling

From section 1.3.1 we know that the outer offset of a model can be generated by computing the Minkowski sum of the model and a ball centered at the origin, and the inner offset can be generated by first computing the Minkowski sum of the (bounded) complement of the model and the ball and then taking its complement (or equivalently the Minkowski difference of the model and the ball, see Proposition 2.3). Since our algorithms compute a voxelized Minkowski sum, its complement can be computed trivially by toggling 0 and 1 for each voxel. Figure 6.1 shows an example of the outer and inner offsets computed using our voxelization algorithms. The algorithm described in chapter 5 is much faster than the one in chapter 4 since the former culls out most surface primitives, but it does not handle holes inside Minkowski sums, so we need to use the algorithm in chapter 4 to compute inner offsets (since the complement of a finite object will contain a hole).

Translational sweeps can also be computed using Minkowski sums (section 1.3.1). Figure 6.2 shows an example of a 3D translational sweep with an error tolerance computed by using the voxelization algorithm described in chapter 5.

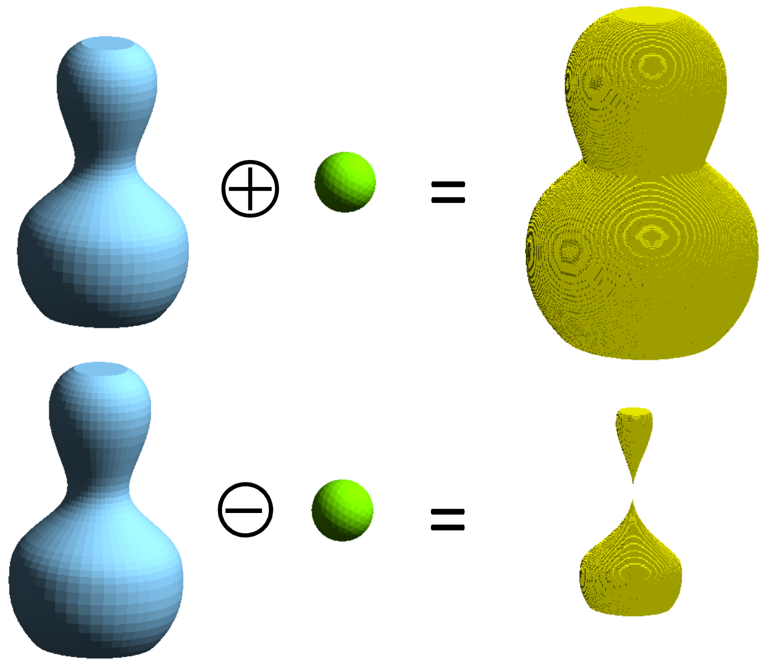


Figure 6.1: The outer (top) and inner (bottom) offsets of a vase model. The vase model has 3,164 triangles and the ball has 500 triangles. The outer offset is computed in 2.52 seconds using the algorithm described in chapter 5. The inner offset is computed in 26.46 seconds using the algorithm described in chapter 4. Both offsets are computed using a resolution of 512^3.

6.2 Motion Planning

In section 1.3.2 we talked about how motion planning problems can be solved by computing C-space obstacles and free C-spaces using Minkowski sums. In this section we give two such examples computed using the voxelization algorithm in chapter 5.

Figure 6.3 shows the C-space obstacle and free C-space of a plug and an outlet. This is a challenging problem since the three prongs of the plug should go into the three corresponding holes of the outlet. Our algorithm successfully found the narrow passageway in the free C-space. Another example of motion planning is shown in Figure 6.4.

6.3 Penetration Depth Computation

From section 1.3.3 we know that the translational penetration depth of two intersecting objects A and B is the same as the shortest distance from the origin to the boundary surface of $B \oplus -A$, and that the vector from the origin to the corresponding closest point gives

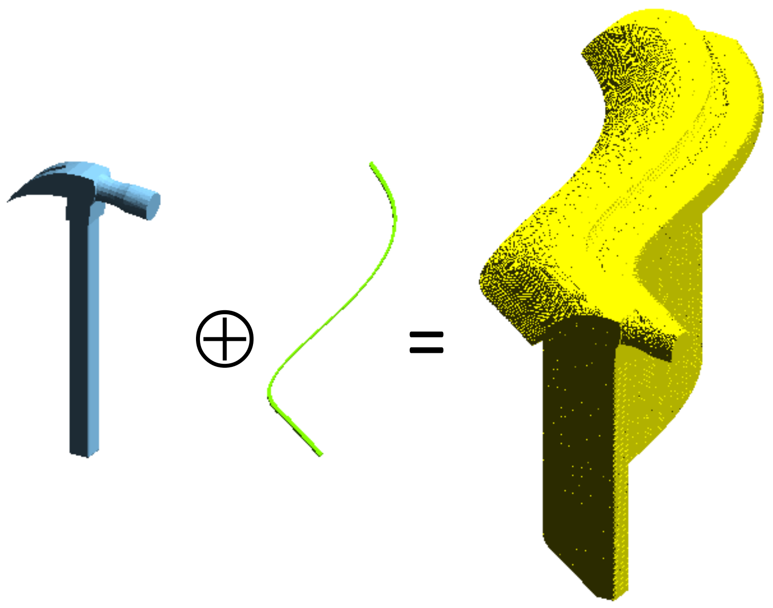


Figure 6.2: Sweep of a hammer model (886 triangles). The sweep trajectory is modeled using a slim polyhedron with 260 triangles that represents the path of the sweep with radius equal to the error tolerance. The voxelized Minkowski sum is computed in 1.77 seconds at a resolution of 1024^3.

the separation direction in which we can translate A away from B. Kim et al. proposed an algorithm for computing penetration depth based on this idea [Kim et al., 2002]. They compute only the Minkowski sum of boundary surfaces and use a depth test to find the closest point on the outer boundary.

We use the surface voxelization of $B \oplus -A$ (discussed in section 5.3.3) to compute the penetration depth. We compute the distance from the origin to all the surface voxels on each slice, and then perform a reduction to find a minimum distance on this slice. Then we perform another pass of reduction on these minimum distances to find the overall minimum distance. Both the reduction and distance computation are implemented using fragment programs on the GPU. Since a fragment program can output a 4-tuple RGBA color, we use the A channel to store the minimum distance and the RGB channels to store the position of the closest voxel. Figure 6.5 shows an example output of our implementation.

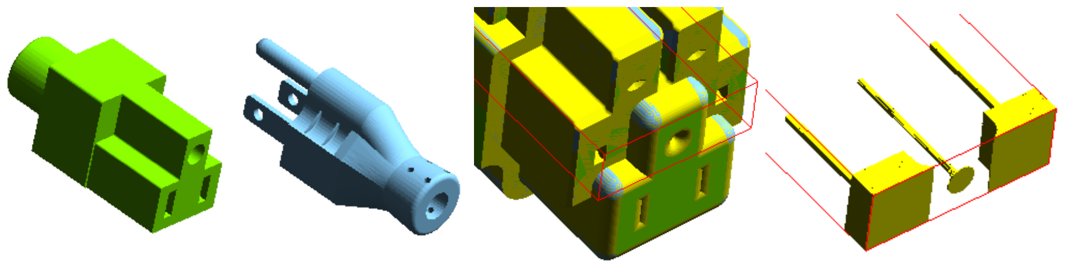


Figure 6.3: Application of our voxelized Minkowski sums in motion planning. From left to right: an outlet (2,018 triangles), a plug (9,262 triangles), a portion of the C-space obstacle outlet $\oplus$ -plug, and the voxelization of the free C-space (the complement of the C-space obstacle) inside the red bounding box. The voxelization is computed in 8.41 seconds at a resolution of 1024^3.

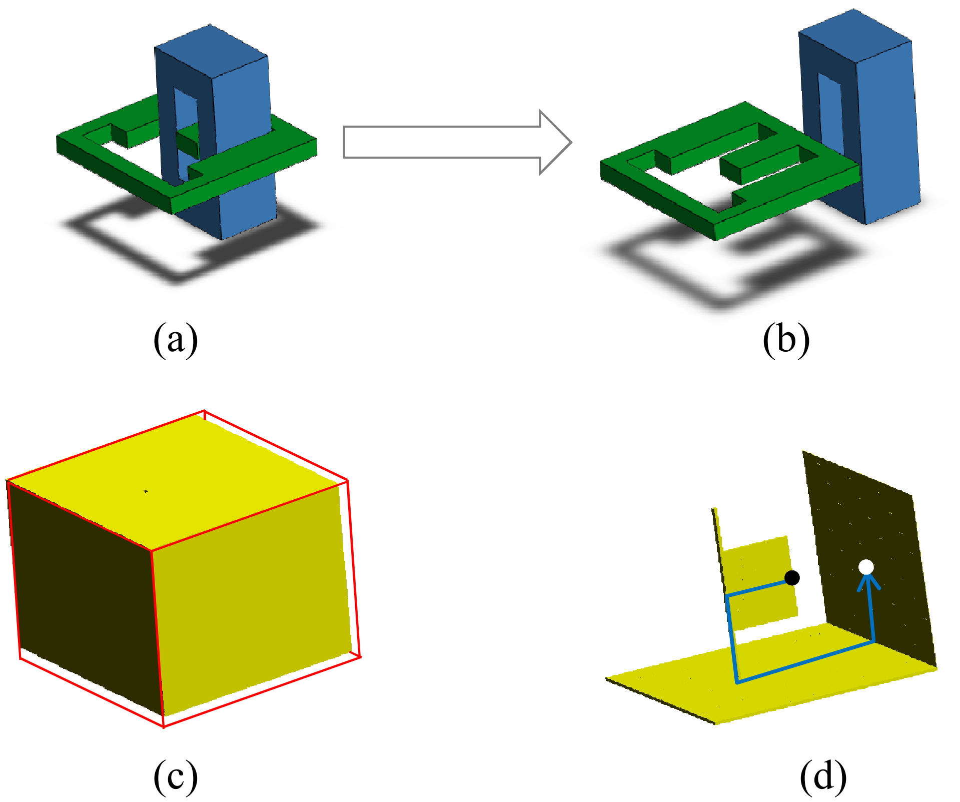


Figure 6.4: (a) The initial configuration. The blue object has 32 triangles and the green one has 64 triangles. (b) The goal configuration. We want to translate the blue object away from the green one. (c) The voxelized C-space obstacle (computed in 2.19 seconds at a resolution of 512^3). Note that there is a hole in the top face. (d) The free C-space inside the red bounding box and a path connecting the initial (black dot) and goal (white dot) configurations.

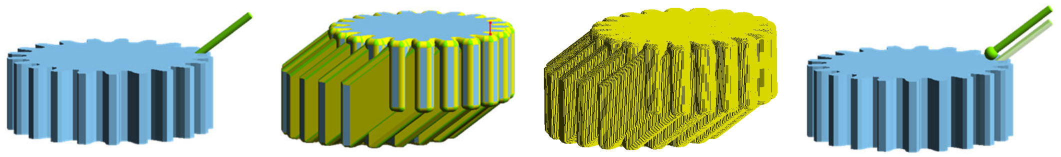


Figure 6.5: Penetration depth between a gear (836 triangles) and a haptic probe (2,498 triangles) that intersects it. The center left figure shows the rendered gear $\oplus$ -probe (the red line connects the origin and the closest point on the boundary surface), and the center right figure shows its voxelization (512^3 resolution). In the rightmost figure, the probe is translated along the computed vector to separate it from the gear. The Minkowski sum is computed in 0.84 second using the voxelization algorithm described in chapter 5 and the penetration depth is found in 0.072 second.

Chapter 7

Conclusions

7.1 Conclusions

In this dissertation, we have presented two algorithms for directly computing a voxelization of the 3D Minkowski sum of two polyhedra on GPUs, without having to first compute a complete boundary representation. Our work is motivated by the fact that most existing algorithms for computing 3D Minkowski sums are B-rep based and suffer from high combinatorial complexity and complex 3D computations such as arrangements and unions, which makes these algorithms both time and memory consuming for inputs with large numbers of facets. By directly computing a voxelization of the Minkowski sum, our approaches avoid such complex computations and are easy to implement.

We have shown that the benefits of our direct voxelization approaches are twofold. First, we can utilize the GPU's built-in rasterization functionality to perform the voxelization, and parallel computing capability to speed up massive geometric data processing (such as the primitive culling and front propagation in chapter 5). Second, voxelized Minkowski sums are more advantageous in applications where a B-rep would need to be sampled and/or point membership classification would need to be performed (such as motion planning, one of the most important applications of Minkowski sums).

Benefits of voxelization are inseparable from two limitations. The first is that we do not compute a boundary representation (although we provide an outer boundary visualization). Since a B-rep is the main input format of most CAD tools and commercial software, our voxelized Minkowski sums can not be directly processed by them. The second is that we compute an approximate Minkowski sum instead of an exact one. The voxelized representation itself is an approximate one, so it can not handle degenerate cases such as lower dimensional features in Minkowski sums. However, we support high resolution of 1024^3, and users can choose the necessary resolution according to the tolerance requirement of the specific application.

The main achievements of this dissertation are summarized as follows:

- We have developed the theoretical background of some important Minkowski sum applications including solid modeling and motion planning, which was previously presented without formal proofs.

- We have analyzed the accessibility and cleanability problem of high pressure waterjet cleaning, to illustrate the relationship between Minkowski sums and the widely used C-space approach for motion planning. We present an approach for finding all the cleanable regions of a polygon under a 2D waterjet cleaning process by means of geometric accessibility analysis that uses Minkowski sums for computing C-spaces.

- We have presented the mathematical formulation and explanation of the commonly used "sweep-along-the-boundary" method for generating Minkowski sums. We introduce a new formula that decomposes the Minkowski sum of two polyhedra as the union of a series of triangular prisms and a translation of each input. Bases on this formula, we describe a GPU-based algorithm to compute a voxelization of the Minkowski sum using a stencil shadow volume technique.

- We have presented a new approach for efficiently computing voxelized Minkowski sums without holes that combines previous convolution-based algorithms for computing Minkowski sums and GPU-based voxelization techniques. We also provide a method to adaptively bound the rounding errors of the primitive culling algorithm to solve the floating point error problem. This voxelization approach is one to two orders of magnitude faster than existing B-rep based algorithms.

7.2 Future Research Directions

We believe that the voxelized Minkowski sum is a promising approach for computing Minkowski sums in 3D or even higher dimensions, given its simplicity and ease of implementation. Its massive parallelism should allow it to automatically benefit from the more and more powerful parallel computing capability of future GPUs and CPUs. Below we suggest some possible future research directions for computing voxelized Minkowski sums, based on the limitations of the two approaches introduced in this dissertation.

- The voxelization algorithm described in chapter 4 is not robust in the presence of floating point rounding errors. These rounding errors may cause small holes inside computed Minkowski sums. The adaptive culling method described in section 5.4 can not be applied here because we can not simply cull out a prism when its orientation test result is within the rounding error bound. One possible solution is to use an exact arithmetic approach such as [Shewchuk, 1997]. However, implementing such exact arithmetic is not trivial and it will unavoidably impact the performance. Finding an efficient way to solve the floating point error problem remains future work.

- Despite its fast speed compared to existing algorithms and also the algorithm presented in chapter 4, the voxelization algorithm described in chapter 5 can not handle holes inside Minkowski sums, which limits its applications. A compromise solution is, if we already know a seed in a hole, we can use the same front propagation technique presented in section 5.3.3 to find all the voxels inside that hole.

- Both the Minkowski sum algorithms presented in this dissertation take polyhedra as their inputs, but in some applications we need to compute the Minkowski sum of two point sets that do not represent meaningful geometric objects. For example in articulated robots, each point represents a series of angles formed by the robotic arms [Curto et al., 2002]. Both algorithms are also limited to Minkowski sums of watertight polyhedra in the 3D space. If we take rotations into consideration, we need to compute 6 dimensional Minkowski sums. We can not handle lower dimensional objects embedded in the 3D space either (e.g., a sweep along a 2D path). One unified approach to overcome the difficulties introduced by both non-polyhedra inputs and higher dimensions is sampling the two inputs into two arrays, converting the Minkowski sum to a convolution of these two arrays, and then computing the convolution using a fast Fourier transform (FFT) [Kavraki, 1995, Curto et al., 2002, Lysenko et al., 2010, 2011]. Similar to our approaches presented in chapter 4 and 5, this FFT approach also computes a voxelization of the Minkowski sum. But they first voxelize (or sample) the two inputs and then compute the Minkowski sum of these two voxelized inputs. Our approaches directly compute the voxelized Minkowski sum from the boundaries of the two inputs. The main drawback of this FFT approach is its low accuracy, because the memory usage limits the resolution used for voxelization (it requires a floating point number for each voxel). Finding a solution that has the advantages of both the FFT and our GPU-based approaches could be promising future work.

Bibliography

Diego Arbelaez, Miguel C. Avila, Adarsh Krishnamurthy, Wei Li, Yusuke Yasui, David Dornfeld, and Sara McMains. Cleanability of mechanical components. In *Proceedings of 2008 NSF Engineering Research and Innovation Conference*, 2008.

Boris Aronov, Micha Sharir, and Boaz Tagansky. The union of convex polyhedra in three dimensions. *SIAM J. Comput.*, 26:1670–1688, December 1997.

Miguel C. Ávila, Corinne Reich-Weiser, David Dornfeld, and Sara McMains. Design and manufacturing for cleanability in high performance cutting. In *Proceeding of 2nd International High Performance Cutting Conference*, 2006.

Miguel Carlos Ávila. *Burr Minimization and Cleanability in High-Volume Manufacturing of Automotive Components*. Ph.D. dissertation, UC Berkeley, 2007.

Hichem Barki, Florence Denis, and Florent Dupont. Contributing vertices-based Minkowski sum of a non-convex polyhedron without fold and a convex polyhedron. In *IEEE International Conference on Shape Modeling and Applications*, pages 73–80, 2009a.

Hichem Barki, Florence Denis, and Florent Dupont. Contributing vertices-based Minkowski sum computation of convex polyhedra. *Computer-Aided Design*, 41(7):525–538, 2009b.

John R. Baumgardner and Paul O. Frederickson. Icosahedral discretization of the two-sphere. *SIAM Journal on Numerical Analysis*, 22(6):1107–1115, 1985.

Evan Behar and Jyh-Ming Lien. Dynamic Minkowski sum of convex shapes. In *Proc. IEEE Int. Conf. Robot. Autom.*, Shanghai, China, May 2011.

K. Berger. Burrs, chips and cleanness of parts - activities and aims in the German automotive industry. In *Presentation at CIRP Working Group on Burr Formation*, 2006.

S.V. Burtsev and Ye.P. Kuzmin. An efficient flood-filling algorithm. *Computers & Graphics*, 17(5):549–561, 1993.

CGAL. CGAL, Computational Geometry Algorithms Library, June 2008. `http://www.cgal.org`.

CGAL. Cgal, Computational Geometry Algorithms Library, 3D Minkowski Sum of Polyhedra, June 2011. `http://www.cgal.org/Manual/3.4/doc_html/cgal_manual/Minkowski_sum_3/Chapter_main.html`.

Bernard Chazelle, David P. Dobkin, Nadia Shouraboura, and Ayellet Tal. Strategies for polyhedral surface decomposition: an experimental study. In *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, SCG '95, pages 297–305, 1995.

Bernard M. Chazelle. Convex decompositions of polyhedra. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 70–79, 1981.

Lin-Lin Chen, Shuo-Yan Chou, and Tony C. Woo. Parting directions for mould and die design. *Computer-Aided Design*, 25(12):762 – 768, 1993.

Byoung K. Choi, Dae H. Kim, and Robert B. Jerard. C-space approach to tool-path generation for die and mould machining. *Computer-Aided Design*, 29(9):657 – 669, 1997.

CUDA. NVIDIA CUDA Programming Guide version 3.0, 2011. `http://www.nvidia.com`.

B. Curto, V. Moreno, and F.J. Blanco. A general method for C-space evaluation and its application to articulated robots. *Robotics and Automation, IEEE Transactions on*, 18 (1):24 – 31, feb 2002.

Direct3D. Direct3D Developer Center, 2011. `http://msdn.microsoft.com/en-us/directx/default`.

Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. Real-time voxelization for complex polygonal models. In *PG '04: Proceedings of Computer Graphics and Applications, 12th Pacific Conference*, pages 43–50, 2004.

Stephen A. Ehmann and Ming C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Computer Graphics Forum*, pages 500–510, 2001.

Elmar Eisemann and Xavier Décoret. Single-pass GPU solid voxelization for real-time applications. In *GI '08: Proceedings of Graphics Interface 2008*, pages 73–80, 2008.

Cass Everitt and Mark J. Kilgard. Practical and robust stenciled shadow volumes for hardware-accelerated rendering. 2002. `http://arxiv.org/pdf/cs/0301002`.

Shiaofen Fang and Hongsheng Chen. Hardware accelerated voxelization. *Computers and Graphics*, 24:433–442, 2000.

Shiaofen Fang and Duoduo Liao. Fast CSG voxelization by frame buffer pixel mapping. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, pages 43–48, 2000.

Randima Fernando and Mark J. Kilgard. *The Cg Tutorial*. 2003.

Efi Fogel and Dan Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. *Computer-Aided Design*, 39(11):929–940, 2007.

Efraim Fogel. *Minkowski Sum Construction and Other Applications of Arrangements of Geodesic Arcs on the Sphere*. Ph.D. dissertation, Tel-Aviv Univ., October 2008.

Pijush K. Ghosh. A unified computational framework for Minkowski operations. *Computers & Graphics*, 17(4):357–378, 1993.

Sarah F. Frisken Gibson. Beyond volume rendering: Visualization, haptic exploration, and physical modeling of voxel-based objects. In *Proc. Eurographics Workshop on Visualization in Scientific Computing*, pages 10–24, 1995.

L. Guibas and R. Seidel. Computing convolutions by reciprocal search. In *SCG '86: Proceedings of the Second Annual Symposium on Computational Geometry*, pages 90–99, 1986.

Peter Hachenberger. Exact Minkowski sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica*, 55(2):329–345, 2009.

Dan Halperin. Robust geometric computing in motion. *Int. J. Rob. Res.*, 21(3):219–232, 2002.

Eugene E. Hartquist, Jai Menon, Krishnan Suresh, Herbert B. Voelcker, and Jovan Zagajac. A computing strategy for applications involving offsets, sweeps, and Minkowski operations. *Computer-Aided Design*, 31(3):175–183, 1999.

Bruno Heidelberger, Matthias Teschner, and Markus H. Gross. Real-time volumetric intersections of deforming objects. In *Vision Modeling and Visualization*, pages 461–468, 2003.

Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2nd edition, 2002.

Evaggelia-Aggeliki Karabassi, Georgios Papaioannou, and Theoharis Theoharis. A fast depth-buffer-based voxelization algorithm. *Journal of Graphics Tools*, 4(4):5–10, 1999.

Anil Kaul and Jarek Rossignac. Solid-interpolating deformations: Construction and animation of PIPS. *Computers & Graphics*, 16(1):107–115, 1992.

L.E. Kavraki. Computation of configuration-space obstacles using the fast Fourier transform. *Robotics and Automation, IEEE Transactions on*, 11(3):408 – 413, jun 1995.

Rahul Khardekar, Greg Burton, and Sara McMains. Finding feasible mold parting directions using graphics hardware. *Computer-Aided Design*, 38(4):327 – 341, 2006.

Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182, 2001.

Young J. Kim, Ming C. Lin, and Dinesh Manocha. Fast penetration depth estimation using rasterization hardware and hierarchical refinement. In *In Fifth International Workshop on Algorithmic Foundations of Robotics*, pages 386–387. ACM Press, 2002.

Young J. Kim, Gokul Varadhan, Ming C. Lin, and Dinesh Manocha. Fast swept volume approximation of complex polyhedral models. In *SM '03: Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, pages 11–22, 2003.

Kevin Kreeger and Arie Kaufman. Mixing translucent polygons with volumes. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, pages 191–198, 1999.

M. C. Leu, P. Meng, E. S. Geskin, and L. Tismeneskiy. Mathematical modeling and experimental verification of stationary waterjet cleaning process. *Journal of Manufacturing Science and Engineering*, 120(3):571–579, 1998.

Duoduo Liao. GPU-accelerated multi-valued solid voxelization by slice functions in real time. In *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, pages 18:1–18:6, 2008.

Duoduo Liao and Shiaofen Fang. Fast volumetric csg modeling using standard graphics system. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications*, pages 204–211, 2002.

Jyh-Ming Lien. Covering Minkowski sum boundary using points with applications. *Comput. Aided Geom. Des.*, 25(8):652–666, 2008a.

Jyh-Ming Lien. Minkowski sums of rotating convex polyhedra. In *Proceedings of the Twenty-fourth Annual Symposium on Computational Geometry*, pages 228–229, 2008b.

Jyh-Ming Lien. Hybrid motion planning using Minkowski sums. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008c.

Jyh-Ming Lien. A simple method for computing Minkowski sum boundary in 3D using collision detection. In *The 8th International Workshop on the Algorithmic Foundations of Robotics*, 2008d.

Jyh-Ming Lien, June 2011. `http://masc.cs.gmu.edu/wiki/SimpleMsum`.

Jyh-Ming Lien and Nancy M. Amato. Approximate convex decomposition of polyhedra. In *Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling*, SPM '07, pages 121–131, 2007.

Min Liu, Yu-Shen Liu, and Karthik Ramani. Computing global visibility maps for regions on the boundaries of polyhedra using Minkowski sums. *Comput. Aided Des.*, 41(9):668–680, 2009.

Ignacio Llamas. Real-time voxelization of triangle meshes on the GPU. In *SIGGRAPH '07: SIGGRAPH Sketches*, page 18, 2007.

T. Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, 1983.

Mikola Lysenko, Saigopal Nelaturi, and Vadim Shapiro. Group morphology with convolution algebras. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 11–22, 2010.

Mikola Lysenko, Vadim Shapiro, and Saigopal Nelaturi. Non-commutative morphology: shapes, filters, and convolutions. *Computer Aided Geometric Design*, In Press, Accepted Manuscript, 2011. doi: DOI:10.1016/j.cagd.2011.06.008.

Naama Mayer, Efi Fogel, and Dan Halperin. Fast and robust retrieval of Minkowski sums of rotating convex polyhedra in 3-space. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 1–10, 2010.

Morgan McGuire, John F. Hughes, Kevin Egan, Mark Kilgard, and Cass Everitt. Fast, practical and robust shadows. Technical report, NVIDIA Corporation, Austin, TX, November 2003. `http://developer.nvidia.com/object/fast_shadow_volumes.html`.

William A. McNeely, Kevin D. Puterbaugh, and James J. Troy. Six degree-of-freedom haptic rendering using voxel sampling. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 401–408, 1999.

J. P. Menon and H. B. Voelcker. Set theoretic properties of ray representations and Minkowski operations on solids. Technical report, The Sibley School of Mechanical Engineering, Cornell University, April 1993.

Heidrun Mühlthaler and Helmut Pottmann. Computing the Minkowski sum of ruled surfaces. *Graph. Models*, 65(6):369–384, 2003.

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

Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9 (2):191–205, April 2003.

OpenGL. The OpenGL Graphics System: A Specification, version 4.0, 2011. `http://www.opengl.org`.

Martin Peternell and Tibor Steiner. Minkowski sum boundary surfaces of 3D-objects. *Graph. Models*, 69(3-4):180–190, 2007.

Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 229–238, 2000.

Joon-Kyung Seong, Myung-Soo Kim, and Kokichi Sugihara. The Minkowski sum of two simple surfaces generated by slope-monotone closed curves. In *Proceedings of Geometric Modeling and Processing – Theory and Applications*, page 33, 2002.

Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

Steven N. Spitz, Antonia J. Spyridi, and Aristides A. G. Requicha. Accessibility analysis for planning of dimensional inspection with coordinate measuring machines. *IEEE Trans. on Robotics and Automation*, 15:714–727, 1998.

Gokul Varadhan and Dinesh Manocha. Accurate Minkowski sum approximation of polyhedral models. *Graph. Models*, 68(4):343–355, 2006.

Gokul Varadhan, Shankar Krishnan, T.V.N. Sriram, and Dinesh Manocha. A simple algorithm for complete motion planning of translating polyhedral robots. *International Journal of Robotics Research*, 25(11):1049–1070, 2006.

Ron Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *ESA'06: Proceedings of European Symposium on Algorithms*, pages 829–840, 2006.

Tony C Woo. Visibility maps and spherical algorithms. *Computer-Aided Design*, 26(1):6 – 16, 1994.

Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng. Conservative voxelization. *Vis. Comput.*, 23(9):783–792, 2007.