Voxelized Minkowski Sum Computation on the GPU with Robust Culling

Wei Li[a], Sara McMains[a]

[a] *University of California, Berkeley*

Abstract

We present a new approach for computing the voxelized Minkowski sum (excluding any enclosed voids) of two polyhedral objects using programmable Graphics Processing Units (GPUs). 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. Finally 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 whole algorithm runs on the GPU and is at least one order of magnitude faster than existing boundary representation (B-rep) based algorithms. It avoids the large number of 3D Boolean operations needed in most existing algorithms and is easy to implement. The voxelized Minkowski sums can be used in a variety of applications including motion planning and penetration depth computation.

Keywords: Minkowski Sums, GPU, Voxelization, Rounding Error Analysis, Path Planning, Penetration Depth

1. Introduction

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

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \tag{1}$$

where a and b denote the coordinate vectors of arbitrary points in A and B, and + denotes vector addition. If A and B represent polygons in $\mathbb{R}^2$ or polyhedra in $\mathbb{R}^3$, $A \oplus B$ can be generated by "sweeping" object A along the boundary of object B (or vice versa). This gives another equivalent definition of Minkowski sums, shown below, where B_a denotes the translation of object B by the vector a.

$$A \oplus B = \bigcup_{a \in A} B_a = \bigcup_{b \in B} A_b \tag{2}$$

A simple 2D example is shown in Figure 1.

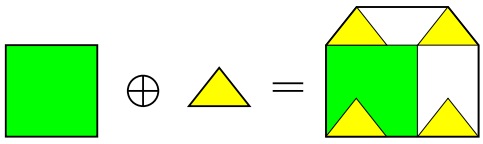

Figure 1: 2D Minkowski sum of a square and a triangle.

Minkowski sums are a fundamental operation for applications such as solid modeling, motion planning, collision detection, penetration depth computation, and mathematical morphology [1, 2, 3, 4]. 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 [5].

Minkowski sums of convex polyhedra can be computed easily and efficiently. Convex hull or Gaussian sphere approaches are commonly used [6, 7]. However, it is much more difficult to compute Minkowski sums of non-convex polyhedra due to the high combinatorial complexity mentioned above. Most existing algorithms for non-convex objects fall into two main categories: convex decomposition [5, 8] or convolution [9, 10, 11]. The first approach decomposes the input non-convex polyhedra into convex pieces, computes all the pairwise Minkowski sums of these convex pieces, and then takes their union. However, the number of pairwise Minkowski sums can be very large (it has quadratic complexity), and computing or even approximating their union robustly is difficult and time-consuming. On the other hand, the convolution-based approach starts 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. The trimming and filtering operations may become very complex since the number of surface primitives also has quadratic complexity and they may intersect each other arbitrarily in 3D space. So both the convex-decomposition and convolution approaches involve many complex 3D computations, and their performance degrades rapidly as the polyhedra complexity increases.

In this paper we present a new approach for computing Minkowski sums of arbitrary polyhedra that extends and improves upon previous work on Minkowski sums and combines these methods with GPU-based voxelization techniques. Unlike most existing algorithms, which compute either an exact [8, 7, 12, 11] or an approximated [10, 5] boundary repre-

sentation, our algorithm aims to directly create both a solid and surface 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 Graphics Processing Unit (GPU) to utilize its rasterization functionality and parallel computation capacity. The benefits of our 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.

Compared with the boundary representation, our volumetric representation of Minkowski sums is more advantageous in various applications such as collision detection, motion planning and penetration depth computation. It provides immediate collision feedback by simply checking if a certain voxel is set to one or zero. Minkowski sum based motion planners often sample the free configuration space to construct a connectivity roadmap [3, 13]; the solid volumetric data provides such sample points with no need of further computation. To find penetration depth, we only need to compute the shortest distance from the origin to all the surface voxels. We will give some application examples of voxelized Minkowski sums in section 5.

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 4.1), 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.

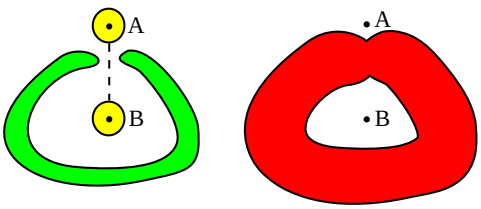

Figure 2: The 2D Minkowski sum (on right in red) of a yellow disk and a green belt contains an enclosed void. The yellow disk can be placed at B, but it cannot go from A to B.

One limitation of our 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 2).

2. Related Work

2.1. Minkowski Sums

Ghosh presented a complete computational framework for Minkowski sums of both convex and non-convex objects where both input objects are represented as "slope diagrams" [6]. The two slope diagrams are merged and their Minkowski sum is recovered from the merged diagram. Unfortunately, the slope diagram operation is complex for non-convex objects; no general implementation of this algorithm is known. Based on a similar idea, Fogel and Halperin proposed computing Minkowski sums of convex polyhedra using a "Cubical Gaussian Map," where geodesic arcs on the Gaussian sphere are projected to the six faces of a bounding cube [7]. This work was later extended by the first author, directly computing the arrangements of geodesic arcs embedded on the Gaussian sphere [14], but also limited to convex objects.

Convex-decomposition based approaches can be categorized into exact and approximate algorithms. 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 [15, 12, 8]. However, these algorithms are limited to relatively simple objects because of their performance. To compute the Minkowski sum of two objects with hundreds of triangles, it usually takes tens of minutes [15]. Varadhan and Manocha proposed another convex-decomposition based algorithm to compute an approximated boundary of Minkowski sums [5]. 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.

Convolution-based approaches have also been proposed for computing the boundary of Minkowski sums. It is well known that 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 [16, 10]. Guibas and Seidel presented an output sensitive algorithm for computing the convolution of 2D curves [17]. Kaul and Rossignac introduced a set of criteria to cull out facets that are not part of the Minkowski sum boundary [2]. These criteria are also used in this paper for Minkowski sum rendering (section 3.1). Peternell and Steiner studied how to extract the outer boundary from the convolution of two objects with piecewise smooth boundaries and compute a local quadratic approximation [10]. Lien proposed to start with a brute force convolution, and compute

facet-facet intersections as 2D arrangements on each facet [9]. Cells from 2D arrangements are then merged and filtered using collision detection tests. Unfortunately the 2D arrangements and collision detection become both time and memory consuming when the size and complexity of the input models increase.

To overcome the computational complexity introduced by 3D operations, some approaches seek to use other lower dimensional representations. Hartquist et al. suggested using "ray representations" (ray-reps) to reduce 3D Minkowski sum computation to 1D Boolean operations [18]. Lien proposed a point-based approach which creates a point set covering the Minkowski sum boundary [19]. Several filters, including the ones introduced in [2], are used to cull out points that are not on the Minkowski sum boundary. Lysenko et al. proposed converting the Minkowski sum to a convolution and computing the convolution using a fast Fourier transform (FFT) [20].

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 [21]. Mühlthaler and Pottmann introduced an explicit parameterization of the convolution of two ruled surfaces [16]. Recently 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 [22].

2.2. GPU-based Voxelization

Voxelization is the process of generating a volumetric representation for geometric objects. In this section we briefly review several GPU-based voxelization algorithms that are related to the techniques we use in this paper. Voxelization algorithms can be classified into *surface voxelization* and *solid voxelization*, depending on whether they voxelize only the boundary surface or the whole interior. Most algorithms described below work for both surface and solid voxelizations. Another classification is *binary voxelization*, where each voxel is represented by 0 or 1, and *non-binary voxelization*, where each voxel is represented by a real value in the range $[0, 1]$. In this paper, we only consider binary voxelizations.

Karabassi et al. presented a depth buffer based voxelization algorithm [23]. The 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. It works only for an object whose boundary can be completely seen from the six orthogonal directions, similarly to [22]. 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 [24]. However, surfaces parallel (or nearly parallel) to the projection direction are not voxelized, and the memory cost for a high resolution volume is high since each voxel requires one byte of memory. To solve these problems Dong et al. proposed projecting the model along three orthogonal directions and encoding multiple voxels in one texel [25].

3. Rendering 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, with applications to motion planning and penetration depth computation that will be discussed later in section 4 and 5, is built upon the rendering results. The rendering algorithm described here can also be used as a standalone 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.

3.1. Surface Primitive Culling

It has been shown in [2] 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 $|F_A| \times |V_B| + |V_A| \times |F_B|$ and $|E_A| \times |E_B|$ respectively. These numbers are as high as millions for two polyhedra with thousands of triangles. 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 3 for a 2D example). We can cull out these primitives and render only the remaining ones. As to be shown in Table 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 3). 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.

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

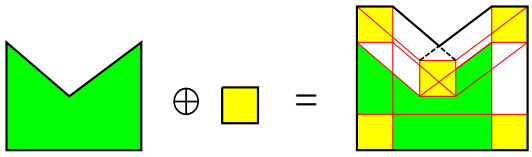

Figure 3: A 2D example of surface primitives in the interior of the Minkowski sum (shown as red lines) and trimmed surface primitives (shown as dashed lines).

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 paper is based on several propositions for primitive culling. The first two propositions (Proposition 1 and 2 below) were first introduced in [2]. However, no proof was provided in that paper. These two propositions were used later, also unproved, in other works [19, 9]. In [26] the authors proved similar propositions, but their criterion for triangle primitive culling is weaker than the one stated below (Proposition 1). Here we use a mathematical description and give proofs of these two propositions.

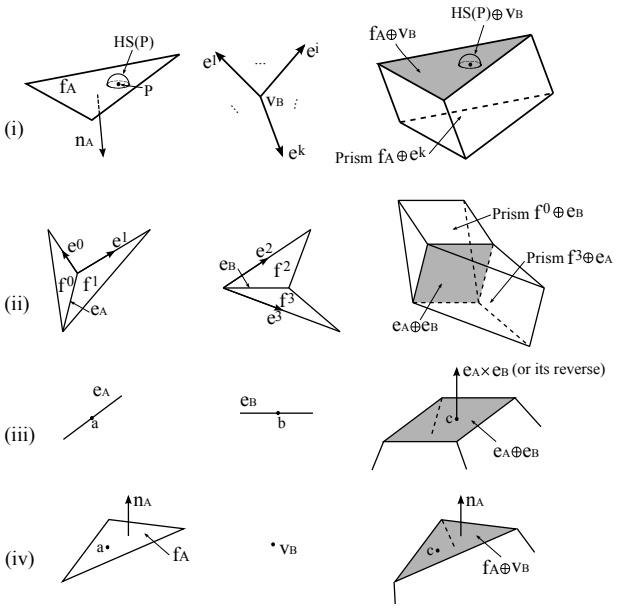

Figure 4: Illustration of the proof of Proposition 1 (i), 2 (ii), 3 (iii), and 4 (iv).

Proposition 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 4 (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 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 4 (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 1 and 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 [6, 7, 22].

The above two propositions only check the relative positions of incident triangles. In this paper we introduce two new propositions that check the orientation of incident triangles. These two propositions are based on the convexity test of vertices and edges. For a vertex, if there exists a supporting plane which 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 [22] the authors considered the case of non-convex edges, but they did not provide a proof.

Proposition 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 4 (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 local maximum or 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 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 4 (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 1 and 4 to cull triangle primitives, and Proposition 2 and 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 1 compares the number of primitives before and after culling for several examples (see Figure 6 for pictures of these input models). We can see that less than 1% of the total primitives remain after culling. Figure 5 shows a Minkowski sum and its triangle and quadrilateral primitives after culling.

3.2. 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 1 and 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 $(e_A \times e_B) \cdot e^i \geq 0$, $\forall e^i$). 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 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 a Quadro FX 5800, which has 30 multiprocessors and a "compute capacity" of 1.3. 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 $0 \leq i, j \leq 15$, 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, 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 [27], 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 average we achieved a three times speedup 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.

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.)

3.3. Rendering Results

We show some results of the above CUDA-based rendering algorithm in Figure 6. The program runs on a Quadro FX 5800 GPU. We also implemented a sequential version of the same algorithm on a Pentium 4 CPU at 3 GHz, and compared the performance between the CUDA and the CPU implementation (see Table 2). Overall the CUDA implementation has a 25 to 30 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 [2]). An example of shape morphing is shown in Figure 7.

4. 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 voxelization algorithms utilize the GPU's rasterization functionality to voxelize boundary surfaces, and then perform a parity check via the stencil buffer [28] or bitwise logic operations [24] to fill

A	B	#tri primitives			#quad primitives			#total primitives
		before culling	after culling	%	before culling	after culling	%	%
bunny	ball	43 M	114 K	0.26%	97 M	54 K	0.06%	0.12%
pig	horse	114 M	694 K	0.61%	255 M	268 K	0.11%	0.26%
Scooby	torus	272 M	268 K	0.10%	612 M	327 K	0.05%	0.07%
dancing kids	octopus	651 M	1,770 K	0.27%	1,466 M	1,300 K	0.09%	0.15%

Table 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.

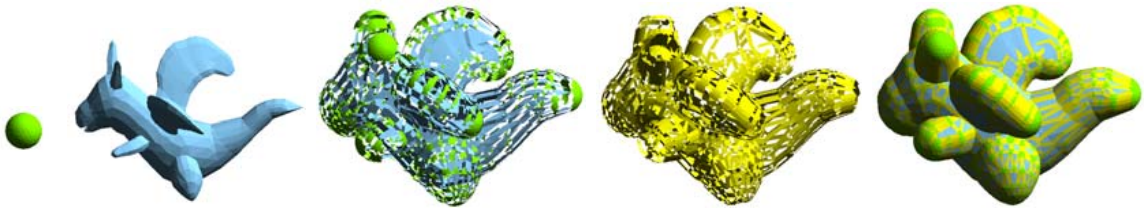

Figure 5: Triangle and quadrilateral primitives after culling. From left to right, each picture represents the two models (ball and dragon), triangle primitives after culling (two different colors represent triangles from the two different models), quadrilateral primitives after culling (yellow), and finally the rendered Minkowski sum.

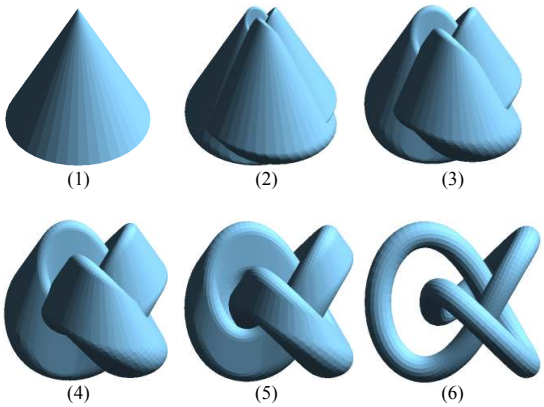

Figure 7: Shape morphing between a cone (78 triangles) and a torus knot (8000 triangles). The animation is computed and rendered at a framerate of 18 frames/second.

Figure 8: 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).

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 [29]. 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 8 gives a 2D illustration of our voxelization algorithm. It consists of three main steps: primitive voxelization, orthogonal fill, and flood fill, as discussed below.

4.1. 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 3.1, 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 8 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 4.3.

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

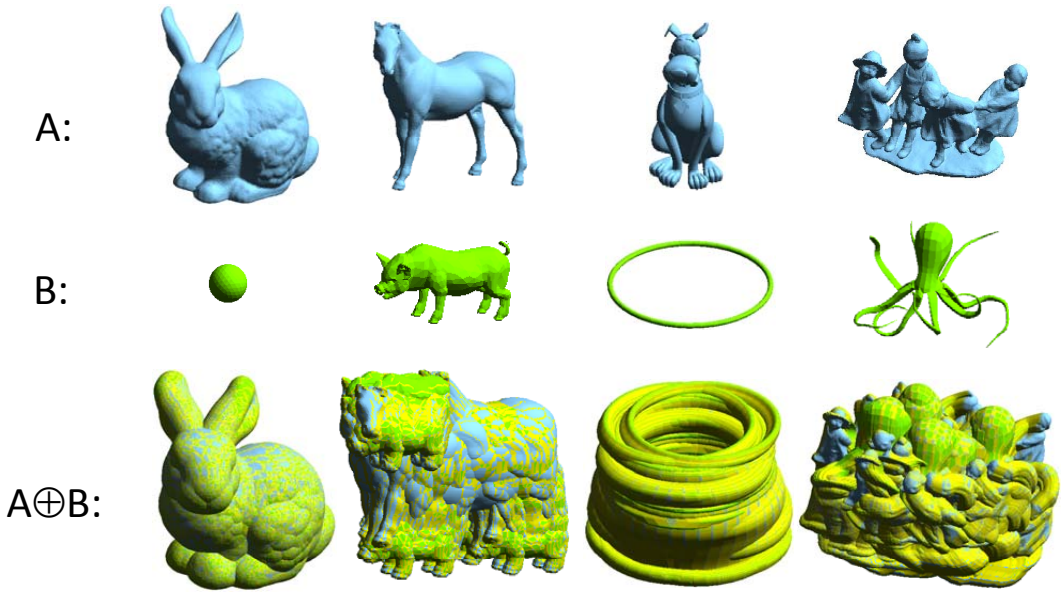

Figure 6: Four examples of CUDA-based Minkowski sum rendering. The Minkowski sum boundary is colored in green, blue, and yellow, representing triangle primitives from the green object, triangle primitives from the blue object, and quadrilateral primitives respectively.

A	B	#tri of A	#tri of B	CUDA time (sec)	CPU time (sec)	Speedup
bunny	ball	86,305	500	1.39	35.97	25.82×
pig	horse	2,784	40,746	3.26	82.00	25.13×
Scooby	torus	170,106	1,600	6.84	198.54	29.03×
dancing kids	octopus	78,706	8,276	15.24	482.15	31.65×

Table 2: Timing for rendering the Minkowski sums in Figure 6 (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.

a corresponding voxel in the 3D volume. In the slicing-based algorithm [24], 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 [25]. 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 [25], 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 computed bounding box by a scale factor of $1 + 2(1 + \epsilon)/[n - 2(1 + \epsilon)]$, where n is the volumetric resolution and ϵ is an infinitesimal number. 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 [30].

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 [25] 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 3). An example of primitive voxelization is shown in Figure 9.

4.2. 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

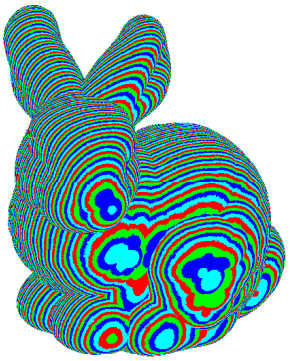

Figure 9: Primitive voxelization (1024^3) of the bunny $\oplus$ ball example in Figure 6. Each voxel is drawn as a point at its center. It is colored in red, green, blue, or cyan if its corresponding bit in the composite 3D texture is in the R, G, B, or A channel respectively. Note that each color band in the figure represents eight slices of voxels.

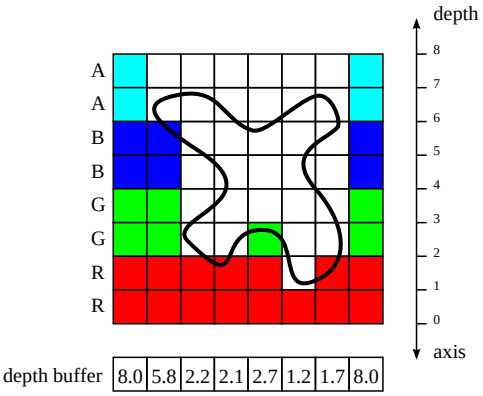

Figure 10: Outer voxels found by orthogonal fill along the specified axis direction. These outer voxels are colored in red, green, blue, or cyan according to their corresponding color channels. Here the volumetric resolution is 8^2 and each color channel has a bit-depth of 2. Note that the depth values, actually from $[0, 1]$, are scaled to $[0, 8]$ for the sake of clarity in the figure.

($+x$, $+y$, $+z$, $-x$, $-y$, and $-z$). An example of such voxels is shown in green in Figure 8. The orthogonal fill is done by rendering the VBO from section 3.2 six times, each time along a different orthogonal direction. These outer voxels serve as seeds for the later flood fill described in section 4.3.

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 10). 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 10, 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 4.1).

4.3. 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 3). 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 8). 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 which 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 [31]. However, this technique relies on being able to perform valid parity checks, which we cannot support because of interior 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 11 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.

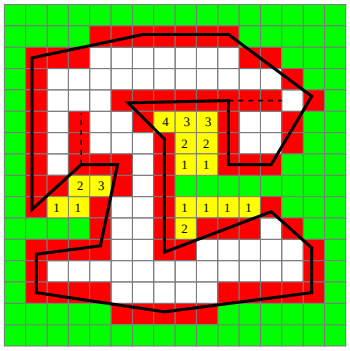

Figure 11: Flood fill of outer voxels. Voxels from primitive voxelization (T_b) are shown in red (we use solid black lines to represent the actual outer boundary of the Minkowski sum and dashed black lines the primitives hidden inside). Outer voxels found by orthogonal fill and not in T_b are shown in 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.

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 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 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 11, 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 512×512×512, it usually takes less than one second for most of the models we have tested (see Table 3). 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 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

4.4. Robust Culling in the Presence of Floating Point Error

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 3.1) relies on a 3D orientation test. For four points a, b, c, and d in $\mathbb{R}^3$, ORIENT3D(a, b, c, d) returns a positive value if a, b, and c appear in clockwise order when viewed from d. To compute ORIENT3D(a, b, c, d), we need to evaluate the sign of a 3×3 matrix determinant [32], as shown below.

$$\text{ORIENT3D}(a, b, c, d) = \begin{vmatrix} a_x - d_x & a_y - d_y & a_z - d_z \\ b_x - d_x & b_y - d_y & b_z - d_z \\ c_x - d_x & c_y - d_y & c_z - d_z \end{vmatrix} \quad (3)$$

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 12 (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.

Figure 12: 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).

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 paper, we compute an upper bound of the rounding error for equation (3). 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.

An upper bound of the rounding error for equation (3) was introduced by Shewchuk [32]. 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 (3) 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 12 (center) shows that some primitives are still missing if we directly apply the upper bound in [32]. 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 (3) with the sum of two original input numbers, as in equation (4), where numbers with subscript 1 or 2 represent coordinates from the first or second input model respectively.

Further analysis indicates that equation (4) 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 (4), 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 (4). To avoid this undesirable subtractive cancellation, we rewrite equation (4) as (5). 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, $q/2 \le p \le 2q$), 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

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \delta), \quad |\delta| \le \epsilon. \tag{6}$$

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 (6). 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 (6) to each of the three operations, we have

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

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

$$\begin{aligned} & \left|fl(x^2 - y^2) - (x^2 - y^2)\right| \\ &= \left|x^2 (\delta_1 + \delta_3 + \delta_1\delta_3) - y^2 (\delta_2 + \delta_3 + \delta_2\delta_3)\right| \\ &\le x^2 |\delta_1 + \delta_3 + \delta_1\delta_3| + y^2 |\delta_2 + \delta_3 + \delta_2\delta_3| \\ &\le (2\epsilon + \epsilon^2)(x^2 + y^2). \end{aligned} \tag{7}$$

Unfortunately, to carry out the rounding error analysis for equation (5), it would be tedious and error-prone to compute the rounding error step by step using model (6), as above, since equation (5) requires 41 floating point operations, 18 more operations than were analyzed in [32]. To simplify the process, we use a convenient notation described in [33, chap. 3]:

$$\langle k \rangle = \prod_{i=1}^{k} (1 + \delta_i), \quad |\delta_i| \le \epsilon. \tag{8}$$

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

$$\begin{aligned} fl(x^2 - y^2) &= \left(x^2 \langle 1 \rangle - y^2 \langle 1 \rangle\right) \langle 1 \rangle \\ &= x^2 \langle 2 \rangle - y^2 \langle 2 \rangle . \end{aligned} \tag{9}$$

$$\textsc{Orient3D}(a,b,c,d) = \begin{vmatrix} (a_{x1}+a_{x2})-(d_{x1}+d_{x2}) & (a_{y1}+a_{y2})-(d_{y1}+d_{y2}) & (a_{z1}+a_{z2})-(d_{z1}+d_{z2}) \\ (b_{x1}+b_{x2})-(d_{x1}+d_{x2}) & (b_{y1}+b_{y2})-(d_{y1}+d_{y2}) & (b_{z1}+b_{z2})-(d_{z1}+d_{z2}) \\ (c_{x1}+c_{x2})-(d_{x1}+d_{x2}) & (c_{y1}+c_{y2})-(d_{y1}+d_{y2}) & (c_{z1}+c_{z2})-(d_{z1}+d_{z2}) \end{vmatrix} \tag{4}$$

$$= \begin{vmatrix} (a_{x1}-d_{x1})+(a_{x2}-d_{x2}) & (a_{y1}-d_{y1})+(a_{y2}-d_{y2}) & (a_{z1}-d_{z1})+(a_{z2}-d_{z2}) \\ (b_{x1}-d_{x1})+(b_{x2}-d_{x2}) & (b_{y1}-d_{y1})+(b_{y2}-d_{y2}) & (b_{z1}-d_{z1})+(b_{z2}-d_{z2}) \\ (c_{x1}-d_{x1})+(c_{x2}-d_{x2}) & (c_{y1}-d_{y1})+(c_{y2}-d_{y2}) & (c_{z1}-d_{z1})+(c_{z2}-d_{z2}) \end{vmatrix} \tag{5}$$

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

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

Note that the two $\langle 2\rangle$s in the first step represent different numbers, i.e., they have different δ_is in equation (8). 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 [33, chap. 3]:

$$|\langle k\rangle - 1| \le 1.01k\epsilon, \quad \text{if } k\epsilon \le 0.01. \tag{10}$$

The condition $k\epsilon \le 0.01$ 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:

$$\begin{aligned} \left|fl(x^2-y^2)-(x^2-y^2)\right| &\le |\langle 2\rangle-1|(x^2+y^2) \\ &\le 2.02\epsilon(x^2+y^2). \end{aligned} \tag{11}$$

Compared to the upper bound in (7), 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) as below:

$$\begin{aligned} |err| \le \; 11.11\epsilon\Big[&\overline{ad}_z\cdot(\overline{bd}_x\cdot\overline{cd}_y+\overline{cd}_x\cdot\overline{bd}_y) \\ &+\overline{bd}_z\cdot(\overline{cd}_x\cdot\overline{ad}_y+\overline{ad}_x\cdot\overline{cd}_y) \\ &+\overline{cd}_z\cdot(\overline{ad}_x\cdot\overline{bd}_y+\overline{bd}_x\cdot\overline{ad}_y)\Big] \end{aligned} \tag{12}$$

where $\overline{ad}_z = |a_{z1} - d_{z1}| + |a_{z2} - d_{z2}|$, etc. A detailed proof is given in Appendix A.

When we perform culling tests, we check the computed determinant of matrix (5) against its error bound computed using the above inequality (12). 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 12 (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 4,times are at most 2% slower with robust culling. The number of remaining primitives after culling increases very little (∼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.

4.5. Voxelization Results and Performance

Figure 13 shows the voxelization results of the four Minkowski sums in Figure 6. The timings under two different resolutions are given in Table 3. Here we use a Quadro FX 5800 GPU with 4 GB video memory. The program runs on CUDA driver 3.0 and 32-bit Windows Vista. 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 ⊕ ball and Scooby ⊕ torus in Figure 6.

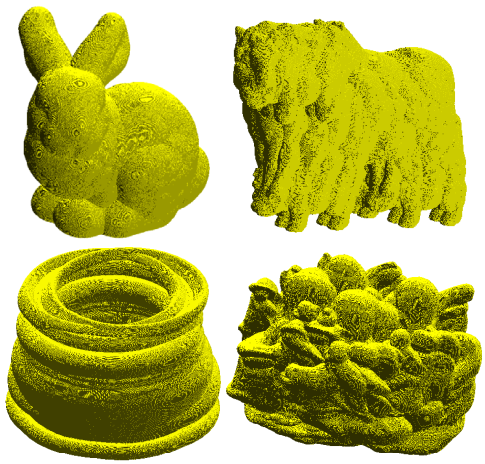

Figure 13: Voxelization (1024^3) of the four Minkowski sums in Figure 6.

Below we compare our voxelization approach with two other recent approaches for approximate Minkowski sum computation, the distance-field based approach [5] and the point based approach [19]. The approach in [5] outputs watertight boundary surfaces that are extracted as isosurfaces from the distance field. The point based approach [19] generates discrete sample points

covering the Minkowski sum boundary. Our approach directly creates surface and solid voxelizations of the Minkowski sum. The accuracy of both the distance-field 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 accuracy of the point based approach is determined by the sampling density. The sampling densities reported in [19] are 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 and our approach do not provide such topological guarantees.

We next compare the performance of our voxelization approach with the method proposed by Lien [9], which, to the best of our knowledge, is the fastest previous implementation for computing general 3D Minkowski sums. We use the same test models as in [9] and report the test results in Table 4, without floating point error checking (since Lien does not check floating point errors). For Lien's method, we use the timings reported in [9] 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 FX 5800 GPU for our timings. From Table 4, 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 about 5,000 triangles [34] in order to match the accuracy of the voxelization at a resolution of 1024^3.

We also found, from the source code Lien kindly provided to us for performance testing, that he also used Proposition 3 and 4 for primitive culling. However, they were not covered in his paper.

5. Applications

The algorithm proposed in this paper can be used in a variety of applications including geometric modeling (e.g., offsetting and sweeping), mathematical morphological operations, and assembly/disassembly. In this section we describe its applications in motion planning and penetration depth computation.

5.1. 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 [1]. Every point in the C-space corresponds to a set of independent parameters that characterize the position and orientation of the rigid object. *Free C-space* is the set of configurations where the object does not collide with the obstacles. The motion planning problem is then reduced to finding a path in the free C-space connecting the initial and goal configurations.

The free C-space is usually computed using Minkowski sums. For P a translating object and Q the union of all the obstacles, the free C-space is the complement of $Q \oplus -P$, where $-P$ is P reflected about the origin. In Figure 14, the free C-space of a plug and an outlet is computed using our voxelization algorithm. 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.

5.2. Penetration Depth Computation

Translational penetration depth is the minimum translational distance to separate two intersecting objects. Mathematically, the penetration depth d_p of two objects A and B is defined as:

$$d_p(A, B) = \inf \left\{ \|\mathbf{d}\| : \mathbf{d} \in \mathbb{R}^3, (A \oplus \mathbf{d}) \cap B = \emptyset \right\}. \qquad (13)$$

Penetration depth is often used in dynamic simulation, haptic rendering, and tolerance verification of CAD models. One can prove that the penetration depth is the same as the shortest distance from the origin to the boundary surface of $B \oplus -A$. The vector from the origin to the corresponding closest point also gives 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 [35]. They compute only the Minkowski sum of boundary surfaces and use a depth test to find the closest point on the outer boundary.

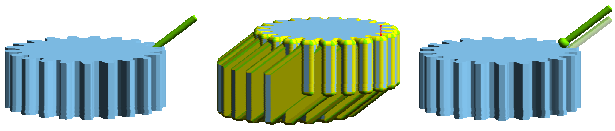

Figure 15: Penetration depth between an intersecting gear (836 triangles) and haptic probe (2498 triangles). The center figure shows gear $\oplus$ -probe. The red line connects the origin and the closest point on the boundary surface. In the right figure, the probe is translated along the computed vector to separate it from the gear. The Minkowski sum is computed in 1.41 seconds and the penetration depth is found in 0.07 second (512^3 resolution).

We use surface voxelization of $B \oplus -A$ 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 final minimum distance. Both the reduction and distance computation are implemented using fragment programs. 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 15 shows an example output of our implementation. Note that since our algorithm only computes outer boundaries, it may give an incorrect result for Minkowski sums with enclosed voids for applications

A ⊕ B	VBO	Prim. Vox.		Ortho. Fill		Flood Fill		#Flood Fill		Total	
		512	1024	512	1024	512	1024	512	1024	512	1024
bunny (86,305) ⊕ ball (500)	1.39	0.43	1.37	0.28	0.70	0.41	2.74	1	4	2.51	6.19
pig (2,784) ⊕ horse (40,746)	3.26	0.15	0.66	0.13	0.55	0.49	5.76	23	111	4.02	10.23
Scooby (170,106) ⊕ torus (1,600)	6.84	0.18	0.68	0.12	0.54	0.78	8.15	44	153	7.92	16.21
dancing kids (78,706) ⊕ octopus (8,276)	15.24	0.30	1.12	0.21	0.63	0.69	8.74	41	185	16.44	25.74

Table 3: Timing for voxelizing the four Minkowski sums in Figure 6 under two different resolutions (in seconds), without floating point error checks. 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. The "512" and "1024" subcolumns represent 512^3 and 1024^3 resolutions.

A	B	#tri of A	#tri of B	Lien's	#Flood Fill		Ours		Speedup	
					512	1024	512	1024	512	1024
inner ear	frame	32,236	96	202.00	161	333	2.25	16.63	90×	12×
bull	frame	12,396	96	289.30	120	240	1.96	13.82	148×	21×
grate1	grate2	540	942	318.50	0	0	1.88	7.66	169×	42×
clutch	knot	2,116	992	347.00	0	0	0.99	4.84	351×	72×
bull	knot	12,396	992	755.10	113	195	2.25	11.52	336×	66×
inner ear	knot	32,236	992	920.80	18	140	2.03	9.70	454×	95×

Table 4: 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.

such as tolerance verification. For haptic rendering, however, the result without enclosed void(s) is actually the desired one for calculating the separation direction.

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.

7. Acknowledgments

We would like to thank NVIDIA for providing us with hardware. Most models are from Professor Lien's website [36] and SolidWorks Content Central [37]. Some are reconstructed using Professor Ju's Polymender to eliminate nonmanifold features [38]. The authors are supported in part by UC Discovery and the National Science Foundation under CAREER Grant No. 0547675 and Grant No. 0621198.

Appendix A. Proof of the Rounding Error Upper Bound Given in Equation (12)

Let the right side of equation (5) be P. If we define

$$\begin{aligned} ad_{z1} &= a_{z1} - d_{z1}, \\ ad_{z2} &= a_{z2} - d_{z2}, \\ ad_z &= ad_{z1} + ad_{z2}, \end{aligned} \tag{A.1}$$

etc., then P can be rewritten as

$$P = \begin{vmatrix} ad_x & ad_y & ad_z \\ bd_x & bd_y & bd_z \\ cd_x & cd_y & cd_z \end{vmatrix}. \tag{A.2}$$

From model (6) and notation (8), we have

$$\begin{aligned} fl(ad_{z1}) &= (a_{z1} - d_{z1})\langle 1\rangle = ad_{z1}\langle 1\rangle \\ fl(ad_{z2}) &= (a_{z2} - d_{z2})\langle 1\rangle = ad_{z2}\langle 1\rangle \\ fl(ad_z) &= \Big(fl(ad_{z1}) + fl(ad_{z2})\Big)\langle 1\rangle \\ &= (ad_{z1}\langle 1\rangle + ad_{z2}\langle 1\rangle)\langle 1\rangle \\ &= ad_{z1}\langle 2\rangle + ad_{z2}\langle 2\rangle . \end{aligned} \tag{A.3}$$

Thus the rounding error of ad_z is

$$\begin{aligned} |err(ad_z)| &= |fl(ad_z) - ad_z| \\ &= |ad_{z1}\langle 2\rangle + ad_{z2}\langle 2\rangle - ad_{z1} - ad_{z2}| \\ &= |ad_{z1}(\langle 2\rangle - 1) + ad_{z2}(\langle 2\rangle - 1)| \\ &\le |\langle 2\rangle - 1| \cdot (|ad_{z1}| + |ad_{z2}|) . \end{aligned} \tag{A.4}$$

If we define

$$\overline{ad}_z = |ad_{z1}| + |ad_{z2}| , \tag{A.5}$$

equation (A.4) can be rewritten as

$$|err(ad_z)| \le |\langle 2\rangle - 1| \cdot \overline{ad}_z. \tag{A.6}$$

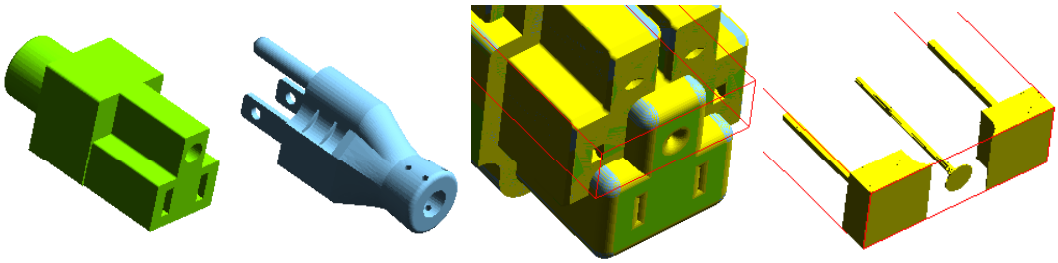

Figure 14: Application of our voxelized Minkowski sums in motion planning. From left to right: an outlet (2018 triangles), a plug (9262 triangles), a portion of the C-space obstacle outlet $\oplus$ -plug, and the voxelization of the free C-space (the complement of the C-space obstacle) inside the red bounding box. The voxelization is computed on a Quadro FX 5800 GPU at a resolution of 1024^3 within 7 seconds.

Note that the two $\langle 2\rangle$s in equation (A.3) represent different numbers, i.e., they have different δ_is in notation (8). For convenience of notation, we do not distinguish the difference between them and rewrite equation (A.3) as

$$\begin{aligned} fl(ad_z) &= (ad_{z1} + ad_{z2})\langle 2\rangle \\ &= ad_z\langle 2\rangle . \end{aligned} \tag{A.7}$$

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 (A.7), we must use $\overline{ad_z}$, as shown in equation (A.6), instead of $|ad_z|$. We will use the same convention in all the following derivations. Similarly, equations (A.6) and (A.7) hold for all the elements in matrix (A.2).

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

$$M_1 = bd_x \cdot cd_y - cd_x \cdot bd_y \tag{A.8}$$
$$M_2 = cd_x \cdot ad_y - ad_x \cdot cd_y \tag{A.9}$$
$$M_3 = ad_x \cdot bd_y - bd_x \cdot ad_y \tag{A.10}$$
$$N_1 = ad_z \cdot M_1 \tag{A.11}$$
$$N_2 = bd_z \cdot M_2 \tag{A.12}$$
$$N_3 = cd_z \cdot M_3. \tag{A.13}$$

Then we have

$$P = N_1 + N_2 + N_3. \tag{A.14}$$

Consider the rounding error of M_1. We have

$$\begin{aligned} & fl(M_1) \\ &= \Big(fl(bd_x)\cdot fl(cd_y)\langle 1\rangle - fl(cd_x)\cdot fl(bd_y)\langle 1\rangle\Big)\langle 1\rangle \\ &= \Big(bd_x\langle 2\rangle \cdot cd_y\langle 2\rangle\langle 1\rangle - cd_x\langle 2\rangle \cdot bd_y\langle 2\rangle\langle 1\rangle\Big)\langle 1\rangle \\ &= \Big(bd_x \cdot cd_y - cd_x \cdot bd_y\Big)\langle 6\rangle \\ &= M_1\langle 6\rangle . \end{aligned} \tag{A.15}$$

The magnitude of the rounding error of M_1 is

$$|err(M_1)| = |fl(M_1) - M_1| \le |\langle 6\rangle - 1| \cdot \overline{M}_1, \tag{A.16}$$

where

$$\overline{M}_1 = \overline{bd}_x \cdot \overline{cd}_y + \overline{cd}_x \cdot \overline{bd}_y. \tag{A.17}$$

Here $\overline{bd}_x$ etc. are defined in the same way as $\overline{ad}_z$ in equation (A.5). Note that the minus sign from equation (A.8) 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 (A.6).

Now consider the rounding error of N_1. We have

$$\begin{aligned} fl(N_1) &= fl(ad_z)\cdot fl(M_1)\langle 1\rangle \\ &= ad_z\langle 2\rangle \cdot M_1\langle 6\rangle\langle 1\rangle \\ &= N_1\langle 9\rangle . \end{aligned} \tag{A.18}$$

If we define

$$\overline{N}_1 = \overline{ad}_z \cdot \overline{M}_1, \tag{A.19}$$

the rounding error of N_1 is

$$|err(N_1)| \le |\langle 9\rangle - 1| \cdot \overline{N}_1. \tag{A.20}$$

Analogous results hold for N_2 and N_3.

Finally we consider the rounding error of P in equation (A.14). 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

$$\begin{aligned} fl(P) &= \Big(\big(fl(N_1) + fl(N_2)\big)\langle 1\rangle + fl(N_3)\Big)\langle 1\rangle \\ &= \Big((N_1\langle 9\rangle + N_2\langle 9\rangle)\langle 1\rangle + N_3\langle 9\rangle\Big)\langle 1\rangle \\ &= N_1\langle 11\rangle + N_2\langle 11\rangle + N_3\langle 10\rangle . \end{aligned} \tag{A.21}$$

To make equation (A.21) 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:

$$\begin{aligned} |err(P)| &= |fl(P) - P| \\ &= |N_1\langle 11\rangle + N_2\langle 11\rangle + N_3\langle 10\rangle - P| \\ &\le |N_1\langle 11\rangle + N_2\langle 11\rangle + N_3\langle 11\rangle - P| \\ &= |P\langle 11\rangle - P| . \end{aligned} \tag{A.22}$$

Now we define

$$\overline{P} = \overline{N}_1 + \overline{N}_2 + \overline{N}_3. \tag{A.23}$$

The rounding error of P is

$$|err(P)| \leq |\langle 11\rangle - 1|\,\overline{P}. \tag{A.24}$$

Using inequality (10), we get

$$|err(P)| \leq 11.11\epsilon\overline{P}. \tag{A.25}$$

This proves the error bound given in equation (12).

References

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

[2] A. Kaul, J. Rossignac, Solid-interpolating deformations: Construction and animation of PIPS, Computers & Graphics 16 (1) (1992) 107–115.

[3] G. Varadhan, S. Krishnan, T. Sriram, D. Manocha, A simple algorithm for complete motion planning of translating polyhedral robots, International Journal of Robotics Research 25 (11) (2006) 1049–1070.

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

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

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

[7] E. Fogel, D. Halperin, Exact and efficient construction of Minkowski sums of convex polyhedra with applications, Computer-Aided Design 39 (11) (2007) 929–940.

[8] P. Hachenberger, Exact Minkowski sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces, Algorithmica 55 (2) (2009) 329–345.

[9] J.-M. 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, 2008.

[10] M. Peternell, T. Steiner, Minkowski sum boundary surfaces of 3D-objects, Graph. Models 69 (3-4) (2007) 180–190.

[11] R. Wein, Exact and efficient construction of planar Minkowski sums using the convolution method, in: ESA'06: Proceedings of the European Symposium on Algorithms, 2006, pp. 829–840.

[12] D. Halperin, Robust geometric computing in motion, Int. J. Rob. Res. 21 (3) (2002) 219–232.

[13] J.-M. Lien, Hybrid motion planning using Minkowski sums, in: Proceedings of Robotics: Science and Systems IV, Zurich, Switzerland, 2008.

[14] E. Fogel, Minkowski sum construction and other applications of arrangements of geodesic arcs on the sphere, Ph.D. dissertation, Tel-Aviv Univ. (October 2008).

[15] CGAL, CGAL, Computational Geometry Algorithms Library, `http://www.cgal.org`.

[16] H. Mühlthaler, H. Pottmann, Computing the Minkowski sum of ruled surfaces, Graph. Models 65 (6) (2003) 369–384.

[17] L. Guibas, R. Seidel, Computing convolutions by reciprocal search, in: SCG '86: Proceedings of the Second Annual Symposium on Computational Geometry, 1986, pp. 90–99.

[18] E. E. Hartquist, J. Menon, K. Suresh, H. B. Voelcker, J. Zagajac, A computing strategy for applications involving offsets, sweeps, and Minkowski operations, Computer-Aided Design 31 (3) (1999) 175–183.

[19] J.-M. Lien, Covering Minkowski sum boundary using points with applications, Comput. Aided Geom. Des. 25 (8) (2008) 652–666.

[20] M. Lysenko, S. Nelaturi, V. Shapiro, Group morphology with convolution algebras, in: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling, 2010, pp. 11–22.

[21] J.-K. Seong, M.-S. Kim, K. Sugihara, The Minkowski sum of two simple surfaces generated by slope-monotone closed curves, in: Proceedings of Geometric Modeling and Processing – Theory and Applications, 2002, p. 33.

[22] H. Barki, F. Denis, F. 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, 2009, pp. 73–80.

[23] E.-A. Karabassi, G. Papaioannou, T. Theoharis, A fast depth-buffer-based voxelization algorithm, Journal of Graphics Tools 4 (4) (1999) 5–10.

[24] S. Fang, H. Chen, Hardware accelerated voxelization, Computers and Graphics 24 (2000) 433–442.

[25] Z. Dong, W. Chen, H. Bao, H. Zhang, Q. Peng, Real-time voxelization for complex polygonal models, in: PG '04: Proceedings of Computer Graphics and Applications, 12th Pacific Conference, 2004, pp. 43–50.

[26] M. Liu, Y.-S. Liu, K. Ramani, Computing global visibility maps for regions on the boundaries of polyhedra using Minkowski sums, Comput. Aided Des. 41 (9) (2009) 668–680.

[27] CUDA, NVIDIA CUDA Programming Guide version 3.0, `http://www.nvidia.com` (2010).

[28] I. Llamas, Real-time voxelization of triangle meshes on the GPU, in: SIGGRAPH '07: SIGGRAPH Sketches, 2007, p. 18.

[29] Y. J. Kim, G. Varadhan, M. C. Lin, D. Manocha, Fast swept volume approximation of complex polyhedral models, in: SM '03: Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications, 2003, pp. 11–22.

[30] R. Fernando, M. J. Kilgard, The Cg Tutorial, 2003.

[31] S. Burtsev, Y. Kuzmin, An efficient flood-filling algorithm, Computers & Graphics 17 (5) (1993) 549–561.

[32] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete & Computational Geometry 18 (3) (1997) 305–363.

[33] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd Edition, SIAM, 2002.

[34] J. R. Baumgardner, P. O. Frederickson, Icosahedral discretization of the two-sphere, SIAM Journal on Numerical Analysis 22 (6) (1985) 1107–1115.

[35] Y. J. Kim, M. C. Lin, D. Manocha, Fast penetration depth estimation using rasterization hardware and hierarchical refinement, in: Fifth International Workshop on Algorithmic Foundations of Robotics, ACM Press, 2002, pp. 386–387.

[36] J.-M. Lien, `http://masc.cs.gmu.edu/wiki/SimpleMsum`.

[37] SolidWorks Content Central, `http://www.3dcontentcentral.com`.

[38] T. Ju, Robust repair of polygonal models, ACM Trans. Graph. 23 (3) (2004) 888–895.