A GPU-Based Voxelization Approach to 3D Minkowski Sum Computation

Wei Li*, Sara McMains
University of California, Berkeley
Berkeley, CA, USA

ABSTRACT

We present a new approach for computing the voxelized Minkowski sum 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. 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 holes 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 for computing Minkowski sums of objects with curved surfaces at similar accuracy. It avoids complex 3D Boolean operations and is easy to implement. The voxelized Minkowski sums can be used in a variety of applications including motion planning and penetration depth computation.

Categories and Subject Descriptors

I.3.3 [Computer Graphics]: Hardware Architecture—*Graphics Processors*; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—*Geometric Algorithms, Languages, and Systems*

Keywords

Minkowski Sums, GPU, Voxelization, Path Planning, Penetration Depth

1. INTRODUCTION

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

$$A \oplus B = \{a + b \mid a \in A, b \in B\} \quad (1)$$

*e-mail: {liw|mc mains}@me.berkeley.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPM 2010 Haifa, Israel

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

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 .

$$A \oplus B = \bigcup_{a \in A} B_a = \bigcup_{b \in B} A_b \quad (2)$$

Minkowski sums are a fundamental operation for applications such as solid modeling, motion planning, collision detection, penetration depth computation, and mathematical morphology [23, 17, 28, 25]. Despite the simplicity of its mathematical definition, computing the Minkowski sums 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 is $O(m^3n^3)$ if they are concave.

Minkowski sums of convex polyhedra can be computed easily and efficiently. Convex hull or Gaussian sphere approaches are commonly used [11, 10]. However, it is much more difficult to compute Minkowski sums of concave polyhedra due to the high combinatorial complexity mentioned above. Most existing algorithms for concave objects fall into two main categories: convex decomposition [29, 13] or convolution [20, 26, 30]. The first approach decomposes the input concave 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. Unlike most existing algorithms, which compute either an exact [13, 10, 14, 30] or an approximated [26, 29] boundary representation,

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 approach 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.

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 [28, 19]. 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, $\sqrt{3}/2/1024$), which is enough for most applications.

Figure 1: The 2D Minkowski sum (on right in red) of a yellow disk and a green belt contains an inner hole. The yellow disk can be placed at B, but it cannot go from A to B.

One limitation of our work is that we don’t compute inner holes of a Minkowski sum, i.e., we only identify its outer boundary. Usually in motion planning, we don’t need to consider inner holes in Minkowski sums, because they represent locations where the object can be placed without collision, but cannot reach from the outside (Figure 1).

2. RELATED WORK

2.1 Minkowski Sums

Ghosh presented a complete computational framework for Minkowski sums of both convex and concave objects where both input objects are represented as “slope diagrams” [11]. The two slope diagrams are merged and their Minkowski sum is recovered from the merged diagram. Unfortunately, the slope diagram operation is complex for concave 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 [10]. This work was later extended by the first author, directly computing the arrangements of geodesic arcs embedded on the Gaussian sphere [9], 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 [4, 14, 13]. 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 [4]. Varadhan and Manocha proposed another convex-decomposition based algorithm to compute an approximated boundary of Minkowski sums [29]. 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. Guibas and Seidel presented an output sensitive algorithm for computing the convolution of 2D curves [12]. Kaul and Rossignac introduced a set of criteria to cull out facets which are not part of the Minkowski sum boundary [17]. 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 [26]. Lien proposed to start with a brute force convolution, and compute facet-facet intersections as 2D arrangements on each facet [20]. 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 [15]. Lien proposed a point-based approach which creates a point set covering the Minkowski sum boundary [18]. Several filters, including the ones introduced in [17], are used to cull out

points that are not on the Minkowski sum boundary.

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 curves [27]. Mühlthaler and Pottmann introduced an explicit parameterization of the convolution of two ruled surfaces [24]. 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 [1].

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 $[0, 1]$. In this paper, we only consider binary voxelizations.

Karabassi et al. presented a depth buffer based voxelization algorithm [16]. 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. 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 [7]. However, surfaces parallel (or nearly parallel) to the projection direction are not voxelized, and the memory cost for 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 [6].

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 the polyhedron interpolation and morphing introduced in [17].

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 [17] 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 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 2 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 (Figure 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 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 following we call a surface primitive *valid* if its intersection with the Minkowski sum boundary has a non-zero area (so a partially trimmed primitive is valid since it has a partial overlap with the Minkowski sum boundary); otherwise we call it *invalid*. Valid primitives will contribute to the boundary of the Minkowski sum, but invalid 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 not valid, 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 [17]. However, no proof was provided in that paper. These two propositions were used later, also unproved, in other works [18, 20]. In [21] 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.

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

#tri of A	#tri of B	#tri primitives			#quad primitives			#total primitives
		before culling	after culling	%	before culling	after culling	%	%
500	2,116	1.1 M	20 K	1.82%	2.4 M	11 K	0.46%	0.89%
500	12,396	6.2 M	42 K	0.68%	13.9 M	27 K	0.19%	0.34%
2,116	12,396	26.3 M	567 K	2.16%	59.0 M	350 K	0.59%	1.08%

Table 1: Examples of surface primitive culling. From left to right, each column respectively shows the number of triangles of 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.

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. 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 $e_i \oplus e_j$.

To achieve coalesced global memory access [5], instead of storing the x, y , and z coordinate of each vertex consecutively $(x_1 y_1 z_1 x_2 y_2 z_2 \dots x_n y_n z_n)$, we store all the x coordinates first, followed by all y , and then all z coordinates $(x_1 x_2 \dots x_n y_1 y_2 \dots y_n z_1 z_2 \dots z_n)$. Since shared memory is much faster than global memory, we copy the 16 triangles and 16 vertices (or 32 edges) from global memory to the shared memory of each block before we perform the culling test. On average we achieved a three times speedup compared to the unoptimized version by using coalesced global memory and shared memory in our implementation.

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.

3.3 Rendering Results

We show some results of the above CUDA-based rendering algorithm in Figure 5. 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.

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 [22] or bitwise logic operations [7] 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 flood fill method directly uses the

adjacency between neighboring voxels. It runs completely on the GPU and avoids the expensive readbacks between GPU and CPU memories.

Figure 6: 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).

Figure 6 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 don’t cull out all the invalid surface primitives. There still exist primitives (or fractions of primitives) hidden inside by the boundary surfaces (see Figure 6 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 onto a 2D plane, usually by orthogonal projection. 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 [7], 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 [6]. The benefits are twofold – it reduces both the memory cost and the number of passes needed to render the object.

Figure 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.

A	B	#tri of A	#tri of B	CUDA time (s)	CPU time (s)	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 of rendering the Minkowski sums in Figure 5 (including primitive culling and VBO generation). From left to right, each column respectively shows model 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.

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 [6], we use a single 3D texture to simplify the voxel access, which has a 32 bit RGBA format and requires only 128MB video memory for the 1024^3 volumetric resolution. By using Multiple Render Targets (MRT) 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 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 bounding box by a factor of $2(1+\epsilon)/[n-2(1+\epsilon)]$, 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 used in [8] for Phong shading.

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 [6] by projecting the input model along three orthogo-

nal 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 primitive voxelization (including three directional projections and texture composition) takes on the order of one second (see Table 3).

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 ($+x$, $+y$, $+z$, $-x$, $-y$, and $-z$). An example of such voxels is shown in green in Figure 6. The orthogonal fill is done by rendering the VBO from section 3.2 six times, each time along a different orthogonal direction and under an appropriate depth test. These outer voxels serve as seeds for the later flood fill 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 7). 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 7, 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 for primitive voxelization (section 4.1).

Figure 7: 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.

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, each taking approximately 1 second for the 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 6). 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 6 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 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 ef-

ficiently on GPUs. What’s 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 [3]. However, this technique relies on valid parity check, 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 don’t 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 8 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 8: 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 , otherwise the flood fill will incorrectly penetrate into the interior of the object. This preprocess can be easily done 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 newly found outer voxels in one 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.

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  $T_o$ 

```

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 8, the final outer boundary surface (solid black lines) consists of primitive voxels (red) which 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 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.

4.4 Voxelization Results and Performance

Figure 9 shows the voxelization results of the four Minkowski sums in Figure 5. The timings under two different resolutions are given in Table 3 for a Quadro FX 5800 GPU. 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 genera-

tion time is nearly proportional to the sizes of 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.

Figure 9: Voxelization (1024^3) of the four Minkowski sums in Figure 5.

We compared the performance of our voxelization approach with the method proposed by Lien [20], 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 [20] and report the test results in Table 4. For Lien’s method, we use the timings reported in [20] for comparison, which were obtained on a PC with two Intel Core 2 CPUs at 2.13 GHz. Since our algorithm runs completely on the GPU, its performance is mainly determined by the GPU (Quadro FX 5800) instead of the CPU. From Table 4, we can see that our approach is at least one order of magnitude faster. Lien’s approach handles internal holes 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 [2] 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) [23]. 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.

A $\oplus$ B	VBO	Prim. Vox.		Ortho. Fill		Flood Fill		#Flood Fill		Total	
		512	1024	512	1024	512	1024	512	1024	512	1024
bunny (86,305) $\oplus$ 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) $\oplus$ 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) $\oplus$ 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) $\oplus$ 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 5 under two different resolutions (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. 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.

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 10, the free C-space of a plug and an outlet is computed using our voxelization algorithm. This is a challenging problem since the three legs 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

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:

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

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 .

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. Figure 11 shows an example of our implementation. Note that since our algorithm only computes outer boundaries, it may give an incorrect result for Minkowski sums with inner holes for applications such as tolerance verification. For haptic rendering, however, the

Figure 11: Penetration depth between an intersecting octopus (8276 triangles) and ball (500 triangles). Center figure shows octopus $\oplus$ -ball. The red line connects the origin and the closest point on the boundary surface. In the right figure, the ball is translated along the computed vector to separate it from the octopus. The Minkowski sum is computed in 10.13 seconds and the penetration depth is found in 0.15 second (1024^3 resolution).

result without inner hole(s) is actually the desired one for calculating 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. Our voxelization approach avoids complex 3D Boolean operations by utilizing 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 hard-

Figure 10: 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.

ware and the anonymous referees for feedback on the presentation. Most models are from Professor Lien's website (<http://masc.cs.gmu.edu/wiki/SimpleMsum>) and SolidWorks Content Central. The authors are supported in part by UC Discovery and the National Science Foundation under CAREER Grant No. 0547675 and Grant No. 0621198.

8. REFERENCES

- [1] H. Barki, F. Denis, and 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*, pages 73–80, 2009.
- [2] J. R. Baumgardner and P. O. Frederickson. Icosahedral discretization of the two-sphere. *SIAM Journal on Numerical Analysis*, 22(6):1107–1115, 1985.
- [3] S. Burtsev and Y. Kuzmin. An efficient flood-filling algorithm. *Computers & Graphics*, 17(5):549–561, 1993.
- [4] CGAL. CGAL, Computational Geometry Algorithms Library, 2010. <http://www.cgal.org>.
- [5] CUDA. NVIDIA CUDA programming guide version 3.0, 2010. <http://www.nvidia.com>.
- [6] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng. Real-time voxelization for complex polygonal models. In *PG '04*, pages 43–50, 2004.
- [7] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers and Graphics*, 24:433–442, 2000.
- [8] R. Fernando and M. J. Kilgard. *The Cg Tutorial*. 2003.
- [9] E. Fogel. *Minkowski Sum Construction and other Applications of Arrangements of Geodesic Arcs on the Sphere*. Ph.D. dissertation, Tel-Aviv Univ., October 2008.
- [10] E. Fogel and D. Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. *Comput. Aided Des.*, 39(11):929–940, 2007.
- [11] P. K. Ghosh. A unified computational framework for Minkowski operations. *Computers & Graphics*, 17(4):357–378, 1993.
- [12] L. Guibas and R. Seidel. Computing convolutions by reciprocal search. In *SCG '86*, pages 90–99, 1986.
- [13] P. Hachenberger. Exact Minkowski sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica*, 55(2):329–345, 2009.
- [14] D. Halperin. Robust geometric computing in motion. *Int. J. Rob. Res.*, 21(3):219–232, 2002.
- [15] E. E. Hartquist, J. Menon, K. Suresh, H. B. Voelcker, and J. Zagaajac. A computing strategy for applications involving offsets, sweeps, and Minkowski operations. *Comput. Aided Des.*, 31(3):175–183, 1999.
- [16] E.-A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *Journal of Graphics Tools*, 4(4):5–10, 1999.
- [17] A. Kaul and J. Rossignac. Solid-interpolating deformations: Construction and animation of PIPS. *Computers & Graphics*, 16(1):107–115, 1992.
- [18] J.-M. Lien. Covering Minkowski sum boundary using points with applications. *Comput. Aided Geom. Des.*, 25(8):652–666, 2008.
- [19] J.-M. Lien. Hybrid motion planning using Minkowski sums. In *Proceedings of Robotics: Science and Systems IV*, Zurich, Switzerland, June 2008.
- [20] J.-M. Lien. A simple method for computing Minkowski sum boundary in 3D using collision detection. In *The 8th Int. Workshop on the Alg. Found. of Robotics*, 2008.
- [21] M. Liu, Y.-S. Liu, and K. Ramani. Computing global visibility maps for regions on the boundaries of polyhedra using Minkowski sums. *Comput. Aided Des.*, 41(9):668–680, 2009.
- [22] I. Llamas. Real-time voxelization of triangle meshes on the GPU. In *SIGGRAPH '07: SIGGRAPH Sketches*, page 18, 2007.
- [23] T. Lozano-Pérez. Spatial planning: a configuration space approach. *IEEE Trans. Comp.*, C-32(2):108–120, 1983.
- [24] H. Mühlthaler and H. Pottmann. Computing the Minkowski sum of ruled surfaces. *Graph. Models*, 65(6):369–384, 2003.
- [25] S. Nelaturi and V. Shapiro. Configuration products in geometric modeling. In *SPM '09*, pages 247–258, 2009.
- [26] M. Peternell and T. Steiner. Minkowski sum boundary surfaces of 3D-objects. *Graph. Models*, 69(3-4):180–190, 2007.
- [27] J.-K. Seong, M.-S. Kim, and 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*, page 33, 2002.
- [28] G. Varadhan, S. Krishnan, T. Sriram, and D. Manocha. A simple algorithm for complete motion planning of translating polyhedral robots. *Int. J. Rob. Res.*, 25(11):1049–1070, 2006.
- [29] G. Varadhan and D. Manocha. Accurate Minkowski sum approximation of polyhedral models. *Graph. Models*, 68(4):343–355, 2006.
- [30] R. Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *ESA '06*, pages 829–840, 2006.