

Finding feasible mold parting directions using graphics hardware

Rahul Khardekar *, Greg Burton, Sara McMains

Mechanical Engineering Department, Etcheverry Hall no. 1740, University of California, Berkeley, CA 94720-1740, USA

Received 19 December 2005; accepted 11 January 2006

Abstract

We present new programmable graphics hardware accelerated algorithms to test the 2-moldability of geometric parts and assist with part redesign. These algorithms efficiently identify and graphically display undercuts as well as minimum and insufficient draft angles. Their running times grow only linearly with respect to the number of facets in the solid model, making them efficient subroutines for our algorithms that test whether a tessellated CAD model can be manufactured in a two-part mold. We have developed and implemented two such algorithms to choose candidate directions to test for 2-moldability using accessibility analysis and Gauss maps. The efficiency of these algorithms lies in the fact that they identify groups of candidate directions such that if any one direction in the group is undercut-free, all are, or if any one is not undercut-free, none are. We examine trade-offs between the algorithms' speed, accuracy, and whether they guarantee that an undercut-free direction will be found for a part if one exists.

© 2006 Elsevier Ltd. All rights reserved.

Keywords: Parting direction; Casting; Injection molding; Graphics hardware; GPU

1. Introduction

In molding and casting manufacturing processes, molten raw material is shaped in molds from which the resulting part must be removed after solidification. Typical rigid, reusable molds consist of two main halves, which are separated in opposite directions (the positive and negative mold 'parting direction') to remove the part. For a given parting direction, the part must be free from undercut features that would make it impossible to define mold halves that could be separated from the part when translated along the positive and negative mold parting directions without colliding with it (see Fig. 1). For small-scale, manual production, one could imagine a worker simultaneously translating and rotating the mold halves along arbitrary paths during removal, but for automated mass production, the two mold halves are typically translated only, always in opposite directions.

We call an object '2-moldable' in a potential mold parting direction if it has no undercuts relative to that direction; the direction is called an undercut-free direction for that object. Formally, an object is 2-moldable in a direction \vec{d} if the

complement of the object can be split into two parts such that one part can be translated to infinity in the direction \vec{d} and the other in the direction $-\vec{d}$ without colliding with the object [1]. Note that if the object is 2-moldable in a direction \vec{d} , then it is also 2-moldable in $-\vec{d}$. For a convex part all directions are undercut-free directions. For other part geometries, there may be no undercut-free direction corresponding to a two-part mold with opposite removal directions; more expensive multi-piece molds with cores and inserts (possibly including threaded inserts) would be required for such parts. In this paper, we present algorithms that address the needs of a CAD user aiming for a part design that can be produced the most economically, in a two-part mold; we leave multi-piece molds to future work.

During the design process, the mold parting direction should be determined before detail design features like the extruded bosses and ribs in Fig. 1 are added. Once this mold parting direction is chosen, there is a need for a real-time tool that will warn the designer as soon as a change is made that makes the part non-2-moldable in the chosen direction. Existing CPU-based algorithms that check for undercuts are too slow to run continuously in the background for complex parts; thus current commercial CAD systems rely on the user to remember to perform the check.

Earlier in the design process, during the conceptual design phase, feedback about whether any undercut-free directions exist for the proposed geometry is helpful. During this stage, identifying multiple undercut-free directions is useful because

* Corresponding author. Tel.: +1 510 918 4288.

E-mail addresses: rahul@me.berkeley.edu (R. Khardekar), greg.burton@gmail.com (G. Burton), mcmains@me.berkeley.edu (S. McMains).

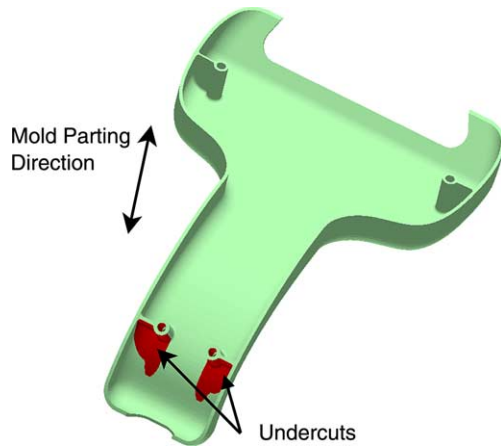


Fig. 1. Rendering with orthographic projection of a part that is not 2-moldable in the indicated direction because of undercut features on the lower cylindrical bosses and their attached strengthening ribs (highlighted in red) (for interpretation of the reference to colour in this legend, the reader is referred to the web version of this article).

a designer can then choose the best possible direction in terms of manufacturing cost and quality. In this paper, we describe efficient graphics-hardware accelerated algorithms to solve the above problems for tessellated input geometry.

2. Background and related work

Recently, commodity graphics hardware has seen enormous improvements in terms of programmability and computational speed. Due to these improvements, graphics processing units (GPUs) have become a viable alternative to CPUs for general purpose computing. Furthermore, ‘Moore’s law’ seems to apply to GPUs but with an even faster improvement rate than for CPUs over the past decade and half: a speedup of roughly 2.4 times a year for GPUs, compared to a 1.7 times speedup per year for CPUs over the same period [34]. If these sustained trends continue, the performance advantage for algorithms that take advantage of graphics hardware will continue to grow. Previous applications of graphics hardware to manufacturing and inspection problems used only the hidden surface removal capabilities of the graphics hardware [2,23,24,29,41,43], but today’s programmable hardware can speed up more complex calculations [15]. The preliminary results we described in [26] and the conference version of this paper [25] were the first application of the programmable capabilities of GPUs to 2-moldability that we are aware of.

Current programmable graphics cards allow general purpose computing in two stages of the graphics rendering pipeline. A vertex processor stage executes a user-defined vertex program in parallel on every vertex passed to the rendering pipeline. A vertex program can change the position, normal, color, and texture coordinates of each vertex. The results of these calculations are passed on to the rasterization stage, where normals, colors, and texture coordinates are interpolated inside the triangles the vertices define. Then a pixel processor executes a user-defined fragment program at every pixel, taking the texture and interpolated vertex data as

input and setting the color and the depth value of that pixel as output.

There is a large body of literature on checking 2-moldability and finding undercut-free directions. In early work in this area, many researchers who considered the problem of finding a casting direction for a two-part mold for a given geometry only look at a limited number of potential parting directions. Ravi and Srinivasan [38] and Wong et al. [46] only consider parting directions along the three principle axes. Chen [8] looks only at the axes of a minimum bounding box. Hui and Tan [22] use a heuristic search approach, which even though not exhaustive, shows significant performance hits on more complex parts with curved surfaces.

Dhaliwal et al. [10] consider all access directions in their algorithm for the automated design of multi-piece sacrificial molds. However, this class of molds is more appropriate for prototyping than for mass production since the molds are destroyed for every part. For their application the problem becomes one of decomposing the mold into machinable pieces rather than de-molding the part. Other researchers have explored automating the design of multi-piece molds and rapid tooling using layered manufacturing [6,7], or shape deposition modeling for sacrificial molds [44].

Calculating visibility is perhaps the most promising approach to finding all 2-moldable directions for parts to be made in permanent molds. Chen et al. introduce the term visibility map to the de-moldability literature in a paper that shows how to minimize the number of cores in parts that cannot necessarily be made in a two-part mold [5]. They find potential undercuts by performing a regularized Boolean subtraction [40] of the part from its convex hull. Woo’s more general paper presents the concept of using convex visibility maps that partition the Gaussian sphere, describing their application to different classes of visibility problems in manufacturing [47]. He relates the degrees of freedom of the surface to be manufactured to the number of manufacturing setups, and shows how clustering of overlapping visibility maps for different surfaces can be used to reduce the total number of setups required for machining. In a subsequent paper Chen and Chou use augmented visibility maps to represent visibility for geometry that does not admit a two-part mold and describe how potential undercuts that cannot be handled by a single core can be subdivided to show a designer what would need to be changed to make a design moldable [4]. A related approach calculates visibility cones [11] and uses them for planning permanent multi-piece molds [36].

A number of papers look to graph-based feature recognition methods to find potential undercuts [17,18,20,51]. Unfortunately graph-based methods break down for interacting features, a shortcoming that [50] address by using a hybrid approach that does not rely exclusively on graph matching. Wuerger and Gadh [48] present an algorithm based on convex hull differences, similar to the [5] approach, to find a parting direction for a two-part mold, but like its predecessors it will not find a parting direction in all cases where one exists. Their more significant contribution is in their companion paper [49], the first we are aware of to describe the implementation of a

discretized representation of a Gauss map. This data structure gives them much better running times than their contemporaries report.

Provably correct algorithms for 2-moldability can be found in the computational geometry literature. Rappaport and Rosenbloom [37] present an $O(n \log n)$ time algorithm (unimplemented) for the 2D case of finding if a polygon is 2-moldable. Ahn et al. combines strong theory with a partial implementation [1]. They show that a definitive answer to whether a polyhedron is 2-moldable in any direction can be obtained via building an arrangement on a sphere as a function of facet normals and orientations where facets may start to obscure each other. Their implementation, however, only tests a heuristically chosen set of directions because of the complexity of implementation and long running time of the complete algorithm. These algorithms all work with tessellated geometry; for curves, [13,32] analyze 2D curved spline input, and for 3D [12] describe an algorithm with both theoretical guarantees and an implementation, but it is limited to C^3 continuous NURBS surfaces only.

3. Checking a direction for undercuts

This section describes our graphics-hardware accelerated algorithms for checking a part when the mold parting direction is given. These algorithms efficiently identify and graphically display undercuts and minimum and insufficient draft angles.

3.1. Undercut detection

For simplicity, we describe our undercut detection algorithm assuming a vertical mold parting direction. We define a part facet as an ‘up-facet’ with respect to a given removal direction \vec{d} if the dot product between the facet’s outward facing surface normal and \vec{d} is greater than zero. For a vertical mold parting direction, take \vec{d} to be the $+z$ -axis. We call the projection of two facets ‘overlapping’ if the interiors of their projections are non-disjoint (note that if the projections touch only at a vertex or along an edge we do not call this ‘overlap’).

Ahn et al. [1] proved that a given part geometry is undercut-free relative to a vertical mold parting direction if and only if it is vertically monotone, i.e. there exists no vertical line that intersects the part interior in more than one disconnected interval. We observe that as a consequence, for a part that is not vertically monotone, vertical lines at the non-vertically-monotone locations will intersect the interior of at least two up-facets. Thus if we project the up-facets of the boundary representation of the part orthographically onto a plane normal to the mold parting direction, the part is undercut-free in this direction if and only if none of the projections of the up-facets overlap.

Kwong observed that this test is simply a visibility test [1,30]. If all the up-facets are completely visible when the object is rendered with orthographic projection, looking down with the eye-point above the object, the object is undercut-free relative to a vertical parting direction. Fig. 2 shows a two dimensional example for a polygon (assume the edges are

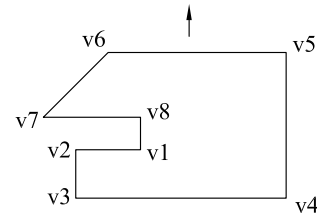


Fig. 2. 2D example for undercut detection.

oriented with normals pointing to the exterior of the polygon). If the part shown is viewed from a point vertically above, the ‘up-edge’ $v1v2$ will be occluded by up-edges $v5v6$ and $v6v7$; thus this contour is not undercut-free in the vertical direction.

Thus determining whether or not a part is undercut-free in a given direction reduces to checking whether there are any partially or completely invisible up-facets when the object is looked upon from the mold removal direction. We solve this visibility problem efficiently with the help of graphics hardware. The inputs to the algorithm are the part geometry and the mold parting direction \vec{d} . The algorithm is as follows:

1. Enable back face culling and standard depth test
2. Set the view matrix parameters
 - a. orthographic projection
 - b. viewing direction $-\vec{d}$
 - c. eye point offset $+d$ from part bounds
 - d. view frustum to encompass part bounds
3. Render the geometry
4. Keep z-buffer but clear frame-buffer
5. Re-render the geometry with depth test set to GL_GREATER
6. Call an occlusion query to test if any pixels were rendered in the second pass

After the first rendering pass (step 3), the z-buffer will hold the distance from the plane of the camera to the visible up-facet for each pixel. During the second pass (step 5), only the (portions of) up-facets that were invisible in the first pass will be rendered. Thus if any pixels are rendered in the second pass, the object is not undercut-free in direction \vec{d} . On recent graphics cards, we can efficiently check if any pixels were rendered in this pass by using their occlusion query functionality, rather than reading back the entire frame-buffer. This makes using the frame-buffer more efficient than using the stencil-buffer, for example. Fig. 3 shows an example of the frame-buffer after step 3 and again after step 5 when the algorithm is run on the part shown with a vertical mold parting direction in Fig. 1.

Our implementation of this algorithm, running on a QuadroFX 3000 GPU, was able to test for the presence of undercuts on parts with over 20,000 facets in less than one millisecond per direction tested using a 256×256 frame-buffer (Fig. 4). Our direction testing rate ranged from 18,200 to 1040 directions per second, on parts with 40–20,676 faces, respectively. Our running times grow only linearly with input size, in contrast to the $O(n \log n)$ growth rate of the Ahn et al. algorithm for testing a direction for 2-moldability (by

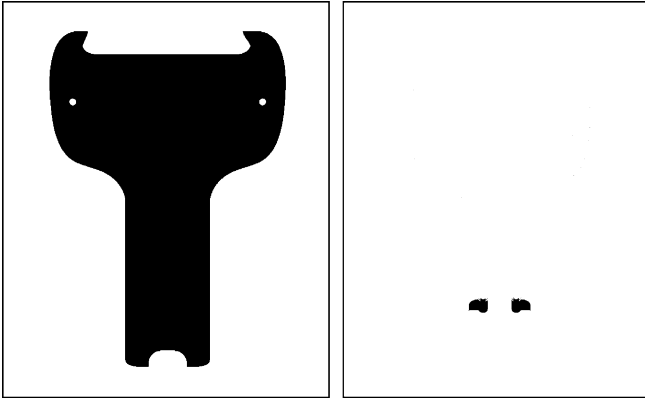


Fig. 3. Screen shots of the frame-buffer showing pixels rendered after the first (left) and second pass (right) of the algorithm.

calculating the object silhouette, projecting it orthographically, and determining if the projected silhouette would be self-intersecting after an epsilon-shrinking operation). The running times for our algorithm on this GPU were over 200 times faster compared to running on the same machine with an older GPU that does not support vertex and fragment programs, so that the CPU (AMD Athlon 1.8 GHz) then had to execute them. Running just the first stage of the Ahn et al. algorithm in software (extracting the silhouette) took five to six times longer than our entire hardware-accelerated algorithm. Faster silhouette extraction might be achieved using more sophisticated sub-linear algorithms [35], but the $O(n \log n)$ plane sweep intersection testing still dominates the theoretical complexity of the Ahn et al. algorithm. (GPU algorithms for silhouette extraction designed for rendering, such as [31], do not seem appropriate for 2-moldability checking due to long vertex programs that slow performance for complex models and the difficulty of accurately performing the epsilon-shrinking and self-intersection tests on a low resolution, rendered silhouette.)

3.2. Highlighting undercut features

Once it is determined that an object is not 2-moldable for a given mold-removal direction, we would like to highlight the

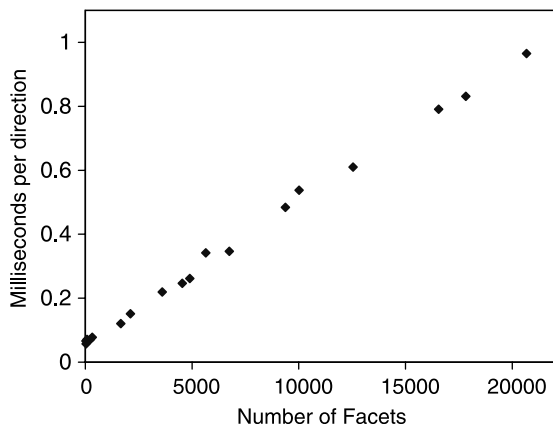


Fig. 4. Running times for 2-moldability checking (QuadroFX 3000 GPU, AMD Athlon 1.8 GHz CPU).

undercuts so that the designer can make the necessary corrections. Up-facet pixels that are rendered in the second pass of the two pass algorithm, along with down-facet pixels that are invisible if the object is viewed from the opposite direction, are the undercuts, which should be highlighted. If the object is illuminated by two light sources located at infinity in the directions \vec{d} and $-\vec{d}$, then the undercut surfaces will be exactly those that are in shadow. For the 2D example in Fig. 2, if the object was illuminated from vertically above and below, the edges $v1v2$, $v8v1$, and portions of $v7v8$ would be in shadow; those edges are the undercut. We can perform this highlighting in real time using the depth texture capability of graphics hardware as detailed below. Depth textures (a.k.a. shadow textures) are textures that store depth values at each pixel location, allowing a second depth test for each pixel as described in [14].

As a preprocessing step before we can display the part with its undercut features highlighted, we perform the following procedure twice, once from the positive mold parting direction and once from the negative mold parting direction. First the scene is projected orthographically with the camera placed above the part, aligned along the positive mold parting direction, and the view direction set towards the object. The z-buffer obtained after this rendering pass is transferred to a depth texture, which will now hold the distance from the plane of the camera to the part for each pixel of the resulting image. We also read back and store the orthogonal viewing matrix associated with this camera position for later use in our vertex program. This procedure is repeated from the opposite mold parting direction.

We can then allow the designer to rotate the object and examine the undercuts in real time, accessing the same two depth textures previously calculated to highlight the currently visible undercuts for any instantaneous viewing direction. We use a vertex program to transform the vertices of each polygon by the two previously generated orthogonal viewing transformations in turn. These two transformed positions are stored as two texture coordinates for each vertex. After interpolation performed during rasterization, the first two components of a fragment's texture coordinate give the location of that pixel in the associated depth texture and the third component gives the depth of the geometry associated with that pixel from the positive (alternately, negative) mold parting direction viewpoint that was used to generate that depth texture. A fragment program checks the depth texture values of both sets of coordinates. If the pixel depth is more than the depth texture value for both the stored textures, then the pixel was occluded by some other geometry from both the positive and negative mold parting directions; we highlight it in red to indicate that it is on a non-2-moldable undercut.

Along with the above computations, the regular lighting computations needed for the scene are also carried out by the vertex and fragment programs. The positions of the lights in the scene are set as constant parameters to the vertex program. We compute the lighting coefficient of the vertices in the vertex program described above and output it in the color attribute of the vertex. In the fragment program, we choose the unlit color

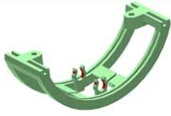

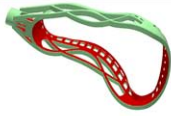
			
No. facets	1,252	38,088	91702
Frame-rate without highlighting	919	315	162
Frame rate with highlighting	330	122	80

Fig. 5. Performance for undercut highlighting on 256 MB NVIDIA GeForceFX 6800 Go card. Models courtesy SolidWorks Inc.

based on the status of the pixel (red if undercut, green otherwise) and then multiply it by the lighting coefficient, which is available from the color attribute of the fragment. We found that faces nearly parallel to the parting direction can give rise to self-shadowing artifacts, where some pixels are mistakenly highlighted as undercuts due to floating point errors. We minimize this problem by using techniques proposed by Reeves et al. [39] and Williams [45] (see Appendix A for details). The parts shown in Fig. 5 were rendered with this process.

3.3. Draft analysis

Although vertical part faces do not constitute undercuts, they make it difficult to remove the part from its mold (primarily due to shrinkage as the part cools). Thus during detail design, ‘draft’ (a slight taper) should be applied to all vertical walls to facilitate mold removal. The angle the modified faces then make with the vertical, typically less than 5°, is called the draft angle. Draft analysis performed today in software can be accomplished more efficiently using vertex programs in hardware.

If the designer specifies a minimum acceptable draft angle, we can use a simple vertex program to highlight facets with insufficient draft. To highlight the facets with draft less than a certain value, we set the mold parting direction and the sine of the threshold draft angle value as constant data for the vertex program. Within the program, we take the dot product of the mold parting direction and the facet normal, and compare it to the stored sine of the threshold draft angle, calculating a rainbow color to apply to triangles with angle less than the desired value. Let r be the ratio of the dot product and the sign of the threshold draft angle. If r is greater than or equal to 1, we color the triangle blue, which indicates that the triangle has a sufficient draft. If r is less than 0.5, we compute the color as follows:

$$\text{color} = (1 - 2r)\text{RED} + (2r)\text{GREEN} \quad (1)$$

If r is greater than or equal to 0.5, we compute the color using the following equation:

$$\text{color} = (2 - 2r)\text{GREEN} + (2r - 1)\text{BLUE} \quad (2)$$

Computing the color and applying it to all triangles reduced the rendering speed from 5 to 50% depending on the geometry. Fig. 6 shows a model of 410,798 triangles colored according to the angle they make with the mold parting direction, which rendered at 21.4 frames per second on a NVIDIA GeForceFX 6800 Go video card and Intel 1.6 GHz Pentium M CPU.

We can also find the minimum draft angle for the entire part in one off-screen rendering pass using a vertex program. The basic idea is to use a vertex program to calculate the sine of the draft angle for each triangle, then render a dummy triangle whose height is set equal to this calculated value instead of the real triangle geometry. The lowest such triangle will be for the smallest sine, corresponding to the minimum draft angle. To implement this approach, when we render the object during this pass, instead of the three actual vertex coordinates of every triangle, we pass in the true triangle normal but dummy vertex position values, $(0,0,z)$, $(1,0,z)$, $(0,1,z)$, respectively. These dummy vertices define a dummy triangle, initially identical for all the facets of the part, which we set to be visible in a small frame-buffer with orthographic projection. In our vertex program, we again set the mold parting direction as constant data, and then take a dot product between the normal of the facet (stored with the vertex) and the mold parting direction, thus calculating the cosine of the angle between the facet normal and the mold parting direction, equivalent to the sine of

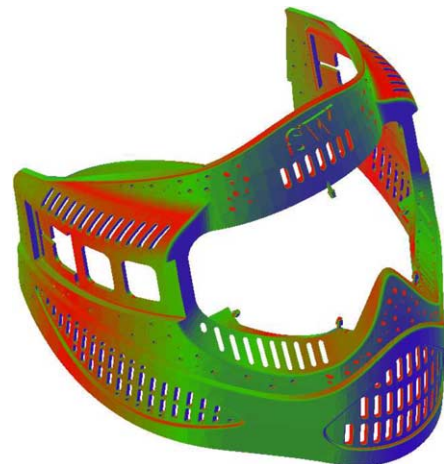


Fig. 6. Model of a mask rendered with the faces colored according to the angle they make with the mold parting direction. Model courtesy SolidWorks, Inc.

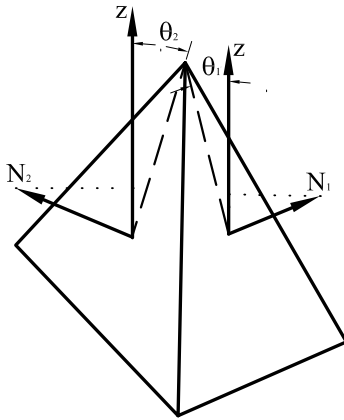


Fig. 7. The dot product of the normal and the mold parting direction (here, $+z$) is the sine of the draft angle θ for a face.

the draft angle (Fig. 7). For the output position value for the dummy vertex, we change the z -coordinate to the sine value calculated. Thus for every input triangle we render a triangle in the frame-buffer with the z -value equal to the sine of the draft angle (Fig. 8). We enable the depth test to `GL_LESS` and set our eye-point on the negative z -axis looking towards the origin with orthographic projection. At the end of the rendering pass only the triangle corresponding to the smallest draft angle will be visible. We then read back the z -value of just one pixel in the interior of the triangle from the frame-buffer and calculate the minimum draft angle, from the arcsine of that value. If we need to identify the triangle that has the minimum draft angle, we assign a unique color to each triangle and read the color back to retrieve the triangle ID.

Fig. 9 shows a graph comparing running times of this GPU algorithm for finding the minimum draft angle to a software implementation. The overhead associated with the GPU algorithm makes it slower for testing very small parts, but for parts with more than a couple thousand facets, the GPU algorithm is faster due to the software algorithm's higher linear growth rate.

During the conceptual design phase, we would also like to provide feedback about potential undercut-free directions so that the designer can choose the direction that is optimal. Knowing the minimum draft angle for different mold parting directions allows us to find the one that maximizes minimum draft, for example. But we only want to compare directions that are actually undercut-free.

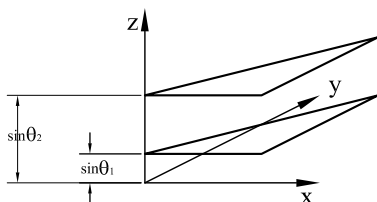


Fig. 8. Dummy triangles rendered for every facet to find the minimum draft angle.

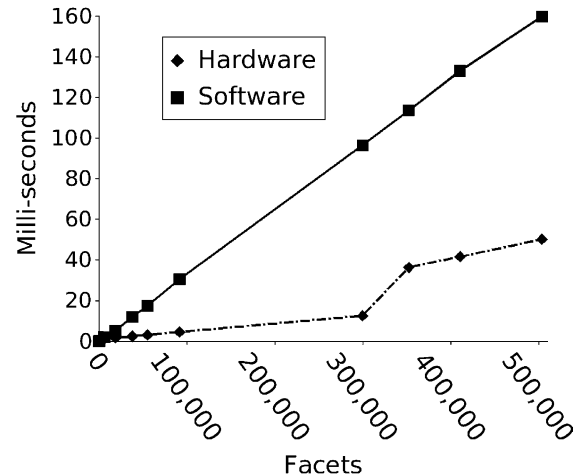


Fig. 9. (a) Performance of minimum draft analysis algorithm on NVIDIA GeForceFX 6800 Go GPU and Intel Pentium 1.6 GHz Mobile CPU.

4. Finding undercut-free directions

In this section we describe two algorithms that we have developed and implemented to choose candidate directions to test for 2-moldability. The key to making these algorithms efficient is to identify groups of candidate directions such that if any one direction in the group is not undercut-free, none are, or if any one is undercut-free, all are.

Both our algorithms make use of a Gaussian sphere, a unit sphere centered at the origin such that every point on it defines a direction in Euclidean 3-space (a unit vector with its tail at the origin and its head on the surface of the sphere). A planar facet defines a great circle on a Gaussian sphere that is perpendicular to the normal vector of the plane. This great circle divides the sphere into two hemispherical regions where the corresponding facet is either always an up-facet or always a down-facet with respect to the set of directions contained in each hemisphere.

4.1. Quadtree algorithm

Our first algorithm runs primarily on the GPU. It is inspired by the theoretical algorithm of Ahn et al. who prove that all combinatorially distinct mold parting directions correspond to 0-, 1-, or 2-cells in an arrangement of great circles on a Gaussian sphere [1]. Every facet normal and normal of the triangle formed by every edge-vertex pair of the part generates a great circle in their arrangement. These great circles correspond to the directions where a part face changes from front-facing to back-facing (directions contained in the plane of the face), and directions where a projection of one part face potentially changes from occluding to not occluding (or vice versa) another part face (directions contained in the planes through an edge-vertex pair from separate triangles).

We observe that there is no need to add the great circles corresponding to face normals to the arrangement, since these are actually a subset of the normals of the triangles defined by edge-vertex pairs. We reduce the number of great circles further by merging adjacent coplanar faces and omitting redundant and non-interacting edge-vertex pairs (those where

neither of the facets adjacent to the edge can be up-facets simultaneously with any of the facets adjacent to the vertex being up-facets).

We project the remaining great circles on a plane tangent to the sphere to obtain an arrangement of lines. We subdivide the line arrangement in a quadtree to obtain 32 lines per quadtree leaf node (because our graphics card has 8 bits for each of the four color channels and we allocate one bit per line; for graphics cards with 32 bits per color channel we would use $32 \times 4 = 128$ lines per quadtree leaf node). We then construct the arrangement in each such leaf node by rendering a half-space for each line with a different color, blending the colors by using the `GL_BLEND` operation (equivalent to a bitwise-or). Fig. 10 shows an arrangement in one quadtree leaf node. Thus each of the up to $32(32 + 1)/2 + 1 = 529$ 2-cells will be rendered in a different color. We select a random sequence of 1024 pixel locations in each leaf node, about twice the maximum possible number of 2-cells, and keep the points having different colors (corresponding to distinct 2-cells in the arrangement). While this certainly does not guarantee that we will find a point in each 2-cell, we found rapidly diminishing returns if we increased the number of initial points tested; for example, doubling the number of points typically only increased the number of cells found by one or two. Since the implementation is only approximate to begin with, we decided to forego the additional overhead. If, while building the quadtree, a cell size falls below a set tolerance limit (less than 1% of the size of the cell we started with) before we have reduced the number of lines (as will always happen if more than 32 lines go through the same point) we stop subdividing and pick 1024 random points in the cell. In either case, we test the directions corresponding to the points, along with face normal and axis directions (which are good heuristic candidates), for 2-moldability.

Fig. 11 shows a part, alongside the directions tested displayed on the Gaussian sphere. Fig. 12 shows timing data for this and additional parts we tested.

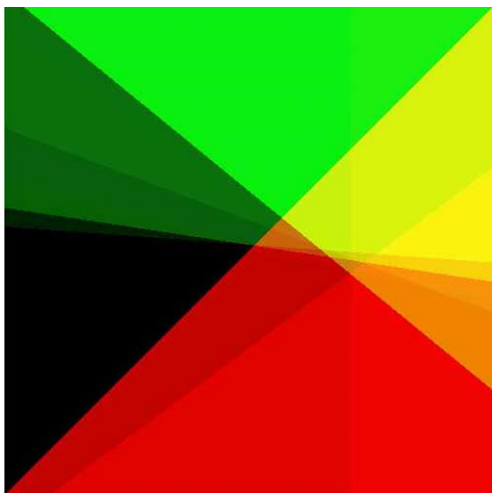


Fig. 10. Arrangement of lines in a quadtree leaf node (for interpretation of the reference to colour in this legend, the reader is referred to the web version of this article).

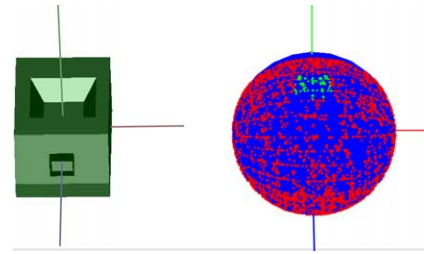


Fig. 11. Sample part (left) with undercut-free directions indicated by the (lighter) green dots on the Gaussian sphere (right). Directions with undercuts are indicated by (darker) red dots (for interpretation of the reference to colour in this legend, the reader is referred to the web version of this article).

Note that in addition to the fact that we cannot guarantee that we will test a direction in the interior of all the 2-cells, this algorithm ignores the 1-cells and 0-cells, which in some cases contain the only undercut-free directions. Furthermore, we found that the speed of frame-buffer read-back was too slow to be useful for practical application on parts with a large number of facets.

4.2. Convex hull intersection algorithm

Our second, more accurate algorithm for finding if undercut-free parting directions exist makes use of convex hulls on the Gaussian sphere. A spherical convex hull (C.H.) of a set of points also on the sphere is a convex spherical polygon bounded by great circular arcs [19]. The set of directions, in which a planar facet A occludes another planar facet B is called the inaccessibility region of B due to A ; it can be calculated exactly and represented as a spherical convex hull on the Gaussian sphere [11]. The inaccessibility regions of A due to B and B due to A lie diametrically opposite to each other on the Gaussian sphere, with the corresponding convex hull vertices projected through the origin.

Recall from Section 3 that an object is not undercut-free in a given direction only if some pair of two up-facets overlap each other when projected orthographically on a plane normal to that direction. Thus only pairs of facets that can potentially become up-facets simultaneously, with their projections also overlapping (non-disjoint interiors) each other, could affect the 2-moldability of the object in any direction. We call such pairs of facets ‘potentially interacting’ facets since they will interact for some, but not all, potential directions. A facet divides space into two half-spaces separated by the plane containing the facet. We call the closed half-space on the side where the facet normal points the positive half-space and the other closed half-space the negative half-space.

A pair of potentially interacting facets will make the object non-2-moldable in the directions lying in the inaccessibility region of the facets due to each other. Taking this into consideration, before calculating convex hulls to find inaccessibility regions of pairs of facets, we can first eliminate pairs that we know can never interact. The following observation allows us to identify all such pairs a priori.





				
No. facets	28	40	80	328
No. great circles	128	258	863	1832
No. directions tested	6,085	17,418	283,904	476,646
No. undercut-free directions found	1	2	34,758	120,299
Time (secs)	2.58	6.89	78	200

Fig. 12. Performance data for quadtree algorithm on NVIDIA QuadroFX 3000 GPU and AMD Athlon 1.8 GHz CPU.

Lemma. *If and only if two facets A and B both lie entirely in one of the half-spaces defined by the plane of the other, and these half-spaces have the same sign, then those two facets cannot interact. In particular if A lies in the positive (alternately, negative) half-space of B and B lies in the positive (alternately, negative) half-space of A , then A and B cannot interact.*

Proof. Consider two facets A and B , each of which lies entirely in the negative half-space of the other (see Fig. 13). Let the great circles on the Gaussian sphere perpendicular to the normal vectors of A and B be called C_A and C_B , respectively. Each great circle divides the sphere into two open hemispheres, within each of which all directions make the corresponding facet either an up-facet (the up-hemisphere of the facet) or a down-facet (the down-hemisphere of the facet). The region within which both the facets are up-facets is the intersection of the up-hemispheres of A and B . Call this region R . We will show that the projections of A and B along any direction in region R cannot overlap.

Call the line of intersection of the two planes containing A and B , respectively L_{AB} . We will show that for projection directions in the region R , the projection of L_{AB} will always separate the projections of A and B ; thus they cannot overlap in this region where they are both up-facets. We know L_{AB} does not overlap A , because A is entirely in the (closed) negative half-space of B , and vice versa, so L_{AB} overlaps neither facet. Because L_{AB} lies in the same plane as A , their projections can never overlap except for projection directions parallel to this plane, the directions on C_A . Similarly the projections of L_{AB} and B cannot overlap except for projection directions on C_B . Thus for projection directions within the up-hemisphere of A , the projection of A will always lie to the same side of the projection of L_{AB} ; it will only cross over the projection of L_{AB} when we move from the up-hemisphere to the down-hemisphere. Likewise for B . Thus, we need only confirm that for some direction in region R the projections of A and B are on opposite sides of the projection of L_{AB} and it will hold true for all directions within region R .

Now consider a vector v that is the average of the facet normals of A and B , as shown in the figure. It is perpendicular to L_{AB} and can be placed on the plane that is the angle bisector between the half planes of A and B , bounded by L_{AB} , that contain the respective facets. This vector lies in the region R on the Gaussian sphere. The projections of A and B along this vector lie on opposite sides of the projection of L_{AB} , since the projections of the respective containing half planes are on opposite sides. Thus the projections of A and B will always be on opposite sides of the projection of L_{AB} for all projection directions in region R , never overlapping in R , so they are not potentially interacting. The case where two facets each lie in the positive half-space of the other is analogous. \square

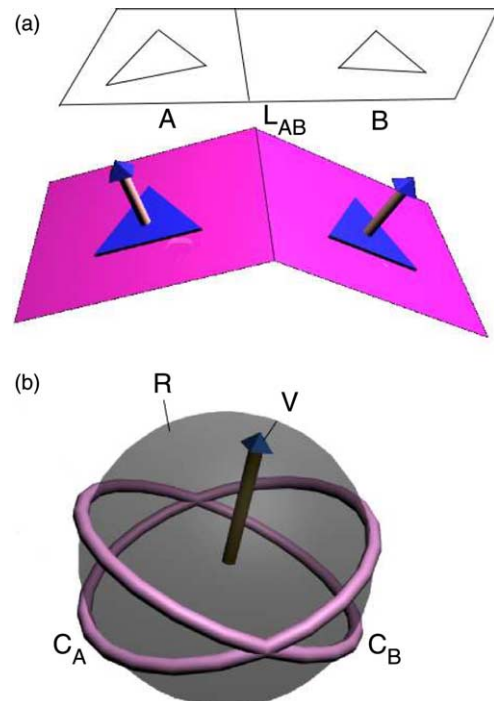


Fig. 13. (a) A pair of non-intersecting triangles (b) Gaussian sphere with great circles corresponding to the pair of triangles.

If two facets do not lie entirely in the same-signed closed half-spaces defined by the plane of the other, then one of them, say facet A , must lie entirely or partially in the open positive half-space of B , whereas B must lie entirely or partially in the open negative half-space of A . If A' is the portion of A in the open positive half-space of B , and B' is the portion of B in the open negative half-space of A , then for any viewing direction vector, call it v , formed by connecting a point in the interior of A' to a point in the interior of B' corresponds to a non-undercut-free parting direction, in which A' occludes B' at this interior point (see Fig. 14). By construction, these two interior points overlap when projected in direction v , so it only remains to show that they are both up-facets. Since B' is in the negative half-space of A , a direction vector leaving any point interior to A and going to any point interior to B' will be pointing towards the negative half-space of A , and therefore v 's dot product with A 's outward facing surface normal is negative, the definition of an up-facet. Similarly at the B' interior point, since the vector is coming from a point in the positive half-space of B , it will have a negative dot product with B 's outward facing surface normal and thus B is also an up-facet for this viewing direction v . Thus for any two facets that do not lie entirely in the same signed half-space relative to each other, there are directions in which they are both up-facets and occlude each other: they are potentially interacting.

Note that for convex objects, which are equivalent to their convex hulls, all the facets are in the negative half-space of all other facets. Thus our test correctly determines that no pair of its facets is potentially interacting; a convex object is 2-moldable in any direction.

Now imagine moving over the surface of the Gaussian sphere, considering different mold parting directions. The only event that changes the 2-moldability of our current direction is when a pair of potentially interacting facets start or stop occluding each other. Thus, the 2-moldability could change only when the current direction crosses one of the arcs bounding the accessibility regions of the potentially interacting facets. These arcs divide the Gaussian sphere into connected regions where the 2-moldability does not change in the interior of any region. Furthermore, if the object is 2-moldable in the directions interior to a region, then it is also 2-moldable in

directions on its boundaries, because arcs that form the boundaries of the inaccessibility regions represent directions where the corresponding projections of the two potentially interacting facets just touch but have zero area overlap. Similarly, if the object is 2-moldable in directions corresponding to the interior points of the arc, it is also 2-moldable in the directions corresponding to the arc 'boundaries,' namely the two vertices of the arc segment. On the other hand, note that the object may be 2-moldable in directions along an arc separating two regions that are not themselves 2-moldable, and it may be 2-moldable in the directions corresponding to the vertices of an arc whose other (interior) points don not correspond to undercut-free directions. Thus to check whether there exists any undercut-free direction for the object, it suffices to test the 2-moldability at the vertices of the connected regions. These vertices will be either the convex hull vertices or the new vertices introduced by the intersection of the original convex hull arcs.

Unlike the previous algorithm, this algorithm is theoretically guaranteed to include an undercut-free direction in the set of candidate directions if any undercut-free direction exists. Of course, in practice, there will be round-off error unless it is implemented using exact arithmetic.

This algorithm can be summarized as follows:

```

for every pair of triangles
  if that pair is potentially interacting
    calculate its inaccessibility region C.H.
    store its bounding great circular arcs
  endif
endfor
calculate intersections of all arcs
test 2-moldability of
  1. C.H. vertices
  2. intersections
    
```

4.2.1. Implementation

We have implemented the above algorithm in C++ on Linux. For every pair of triangular facets in the file, we determine if that pair is potentially interacting by checking whether they are coplanar or in the same signed half-space relative to each other as previously described. On average this reduced the number of triangle pairs by about 75% in the parts we tested (see Fig. 17, for which examples 71–90% of the potentially interacting pairs were eliminated from further testing). Unfortunately, the number of potentially intersecting pairs is still $O(n^2)$, for a potential total of $O(n^4)$ intersections to test, but since we are intersecting only segments, not whole lines, it is unlikely we would see that many intersections in practice.

If a pair of facets is potentially interacting, we next find the inaccessibility region of the two facets. This region is a spherical convex hull of points on the Gaussian sphere defined by the set of nine vectors created from connecting each vertex from one facet to each vertex in the other facet [11]. (In the cases where the facets share a vertex, we drop the null vector

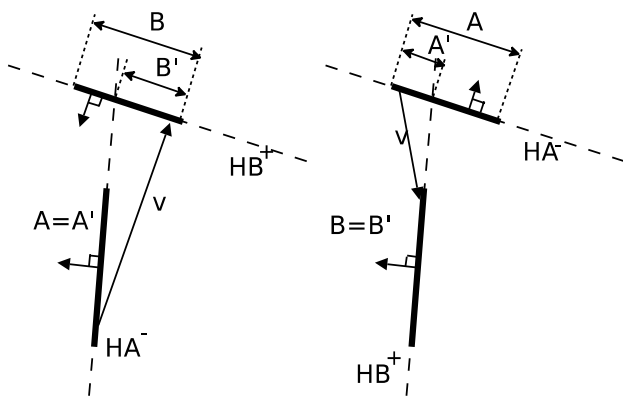


Fig. 14. Two examples of facets A and B , which do not lie entirely in the same signed half-spaces defined by the plane of the other.

from further computation.) Again, we normalize the vectors and place their tails at the origin in the center of a sphere. We then project the vectors onto points on a plane placed tangent to the sphere at the pole of a hemisphere containing the nine vectors, so that their spherical convex hull projects with no overlap. The vectors will always be co-hemispherical assuming our original object boundary was not self-intersecting (see Appendix B for the proof). In practice, we found that simply calculating the average of the vectors was sufficient for choosing a tangent plane orientation with an associated hemisphere that contained all the vectors for all of the parts we tested. On the projection plane, we calculate a standard 2D convex hull, the vertices of which we project back to the sphere and connect with great circular arcs with the same connectivity, giving us the spherical convex hull.

We tried three different convex hull calculation approaches. The first was the gift-wrapping convex hull algorithm as described in [9]. Since we are always using sets of only nine points, a simple, brute force approach like gift-wrapping would seem adequate. However, for pairs of nearly coplanar facets, the projections of the nine vectors from one facet's vertices to the other's vertices will be nearly co-great-circular. Thus their projections onto the plane will be nearly collinear. Gift-wrapping, which works by comparing the signs of the cross products of vectors between candidate convex hull points, can become unstable when points are very close to collinear. (The issue is with numerical imprecision that changes the sign of the cross products, since the sign indicates which side of another line a point is on. This qualitative error answering a sidedness query can lead to mutually impossible results when calculating the cross products with different subsets of the collinear points, leading to an infinite loop or a self-intersecting convex hull.) Given that the number of these cases was large in a significant number of our test geometries, we next turned to exact arithmetic.

We interfaced with two convex hull algorithm implementations from the Computational Geometry Algorithms Library (CGAL) [3], Bykat's non-recursive version of quickhull as well as Akl and Toussaint's convex hull algorithm. We used the Cartesian kernel with the MP_Float (multi-precision float) number type, which can represent floats with arbitrary precision and uses exact arithmetic for numerical operations. As expected, these implementations were much slower due to the overhead of exact arithmetic, and we found to our surprise that for several of the sets of nine input points we generated they ran out of memory (using 1 GB of RAM and 2 GB of swap space), again due to sets of nearly collinear points.

We had the best luck when we implemented Graham's scan algorithm as described in [33] to calculate the convex hull. It sorts the input points by angle from a pivot, deleting from the list those points with equal angle but smaller radius from the pivot than another point in the set, and builds the hull based on the final sorted list. Graham's scan is therefore well suited to handle data sets with collinear input points, because small quantitative round-off errors calculating the angles will only cause small quantitative errors in the results, not mutually impossible answers to sidedness queries.

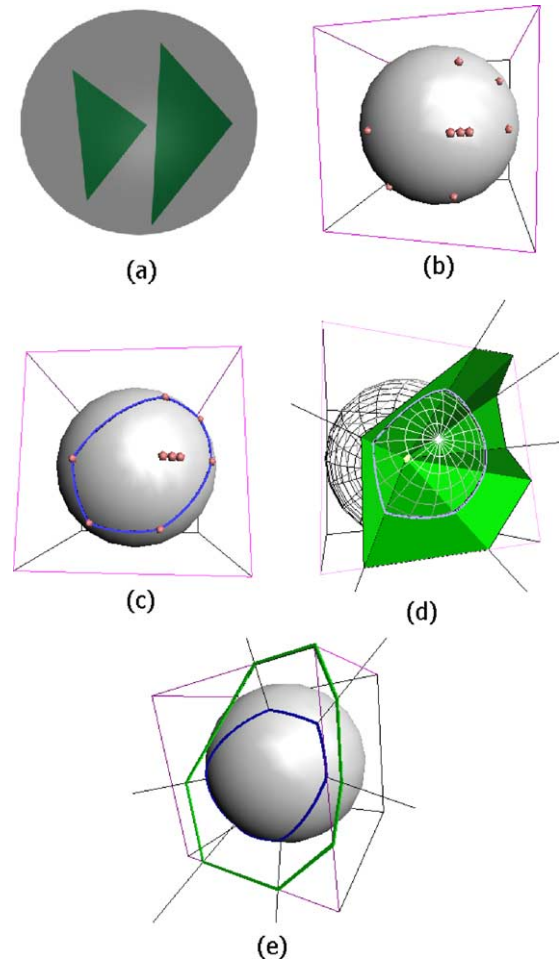


Fig. 15. (a) A pair of triangles (b) Points on the Gaussian sphere obtained from the pair of triangles (c) Spherical convex hull obtained on the sphere (d) Projecting the convex hull on a circumscribing cube (e) The convex hull obtained on the faces of the cube after splitting.

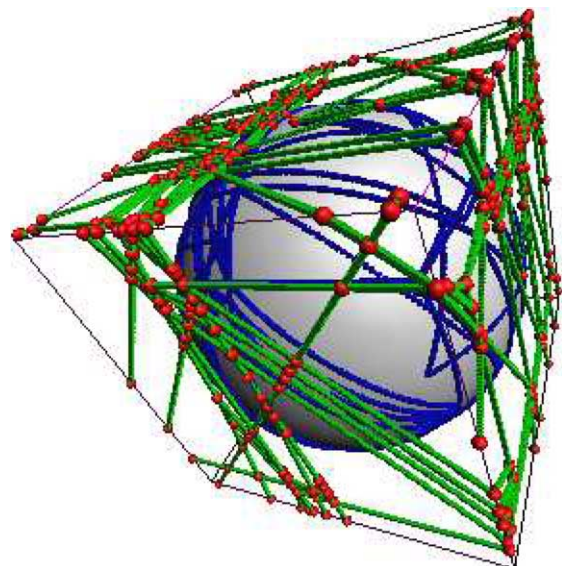


Fig. 16. Points of intersections of lines on the hemi-cube faces.




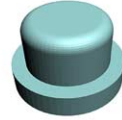

					
No. facets	28	80	328	1994	4634
No. pairs	378	3,160	53,628	1,987,021	10,734,661
No. potentially interacting pairs	140	749	15,228	202,050	3,089,765
Time (secs) C.H. calculations	.01	.04	.78	12.37	184.35
Time (secs) 2-moldability testing	<.01	<.01	<.01	.09	582.16
No. directions tested	1	7	26	358	1,470,416

Fig. 17. Running times for our Convex Hull Intersection Algorithm to find an undercut-free direction (NVIDIA QuadroFX 3000 GPU and AMD Athlon 1.8 GHz CPU).

We verified that from every pair of triangles we get two diametrically opposite inaccessibility regions. Thus, when all the convex hulls are added on the sphere, the arrangement obtained is symmetric about the center. Thus, for further computation it is sufficient to consider the portions of the arc segments that are in any one hemisphere.

Again, it is simpler to calculate great-circular arc intersections if we project them to a plane, where this time the problem maps to an intersection of straight line segments. To avoid points projecting to infinity and to make more uniform use of the floating point precision, we actually project onto a circumscribing, axis-aligned hemi-cube, specifically, the cube faces corresponding to the $x=1$, $y=1$ and $z=1$ planes. We project each convex hull to these hemi-cube faces, splitting segments across the faces if they project to more than one, and clipping at the boundaries. Thus for every face we obtain a set of straight line segments. Fig. 15 shows this process for a pair of triangles.

We first check the vertices of these lines to see if they represent undercut-free directions. If none of the vertices are 2-moldable, then we find the intersections of the lines and check those intersections for 2-moldability (Fig. 16). If we are only interested in finding whether the object is 2-moldable, we can stop after we find an undercut-free direction, otherwise we can continue until all the line vertices and their intersections are checked. Again, we initially tried CGAL for calculating the intersections, but found that we ran out of memory for more complex inputs (presumably the large number of overlapping

and collinear-up-to-our-limited-numerical-precision arc segments were the problem), so we reverted to faster floating point arithmetic intersection calculations.

Details of running times to find an undercut-free direction for different parts without checking heuristic directions first are given in Fig. 17. The convex hull calculations for the more complex parts take from seconds to minutes; 2-moldability testing time depends on how many directions we actually end up checking before finding one that is feasible. The final part in the table was the most challenging with only one undercut-free direction, which our algorithm only found after about fifteen minutes of processing.

5. Future work

Our original proof-of-concept implementation of the convex hull intersection algorithm could be further optimized. Currently we test all the vertices we generate, even those that lie in the interior of an inaccessibility region of another pair of facets. Because the inaccessibility regions are regions of non-2-moldability, such vertices will always be non-2-moldable. The question is whether the time it takes to identify that a vertex is interior to another convex hull is less than the time we can save by not checking the 2-moldability of such vertices. A promising approach could be to find the (non-regularized) union of the interiors of the non-2-moldable regions and only test vertices of this union. An alternate approach to reduce the running time for geometries that are approximations to curved

surfaces would be to look at the original control polygon geometry.

Another area of interest for future work is multi-piece molds with cores and inserts. Our algorithm for checking for 2-moldability can be easily extended to consider additional removal directions for side pulls if the designer specifies the directions. If no removal direction guidance is given, the problem of finding a mold design that is guaranteed to minimize the number of mold pieces appears to be NP-hard [36]. But we believe that GPU-based algorithms can be used with heuristic approaches to facilitate the design of parts to be manufactured in multi-piece molds as well. For example, GPU algorithms could be developed to calculate the decision criteria discussed in [38] in order to assist designers in choosing a (primary) parting direction.

6. Conclusion

This paper presents two algorithms that use an efficient new GPU undercut detection algorithm as a subroutine for detecting if any undercut-free parting directions exist for a faceted part geometry. We introduce a new criteria for identifying which pairs of facets are potentially interacting that we prove eliminates all pairs of facets that can never occlude each other. This optimization typically removes 70–90% of facet pairs from further processing, although unfortunately it does not reduce the worst-case theoretical complexity. For a predetermined parting direction, by exploiting the capabilities of programmable graphics hardware, we are able to achieve sub-millisecond undercut detection and real-time undercut highlighting and draft analysis.

Acknowledgements

NVIDIA donated a graphics card used in this work. The authors were supported in part by UC MICRO and the Hellman Foundation.

Appendix A

We use two techniques to minimize artifacts generated during highlighting undercuts. The first technique addresses errors that arise due to a sampling mismatch between the viewpoints. While carrying out the depth test between a pixel's interpolated depth value and its corresponding depth texture value, we also check the four adjacent texels in the depth textures and compare their depth values to the depth value of the pixel. If any of the depth comparison passes, then we set the pixel as 2-moldable. First proposed by [39], this process of simultaneously testing the adjoining texel is now implemented in hardware and can be accessed via the `GL_ARB_shadow` OpenGL extension. The second technique uses polygon offsetting, first introduced by [45] and now available in OpenGL, which offsets the z value of every pixel before it is written in the z -buffer. The offset amount is calculated as $factor \times DZ + r \times units$ where $factor$ and $units$ are user-defined. DZ is the maximum depth slope (change in z) of the polygon

within that pixel and r is the smallest resolvable z -buffer resolution. We refer the reader to the OpenGL Programming Guide for further details about polygon offsets [42]. While rendering the scene to the depth texture from the $+z$ and $-z$ directions, we enable polygon offsetting to displace the surfaces away from the viewpoint. This reduces artifacts due to limited floating point precision and the texture resolution. We found that using either the standard values of $factor=1$ and $units=1$ or the more aggressive values of $factor=1.1$ and $units=4$ recommended by Mark Kilgard of NVIDIA [27] worked well on NVIDIA cards.

Appendix B

Theorem. For any two triangles in a non-self-intersecting triangulated boundary representation of a 2-manifold part, the nine vectors joining the vertices of one triangle to the vertices of the other correspond to nine co-hemispherical points on the Gaussian sphere.

Proof. We will prove this theorem by proving that for a pair of such triangles, there always exists a separating plane such that the interiors of the triangles lie on opposite sides of the plane. (If the two triangles share an edge, then the edge lies on the separating plane and the interiors lie on the opposite sides of the plane.) Once we prove that there exists such a separating plane, we will show that the vectors joining vertices of one triangle to those of the other will all be in one closed hemisphere of the Gaussian sphere defined by this separating plane. \square

Lemma. For any two triangles whose interiors are disjoint, there exists a separating plane such that the triangle interiors lie on the opposite side of the plane.

Proof. The above lemma is an extension of the separating axis theorem, which states that there exists a separating plane for any two triangles that are *completely* disjoint. Our proof follows the same outline as the proof of the separating axis theorem as provided in [21].

First, we will review a few properties from computational geometry that we will use in the proof. A Gauss map of a convex polytope is a partition of the Gaussian sphere according to the rule: a point v on the sphere is associated with the polytope feature extremal in direction v . A feature is extremal in direction v if it is at least as far in direction v as any other part of the polytope. For a convex polytope, vertices map to convex regions on the Gauss map; edges map to great circles where the great circles represent directions normal to the edges; and each face maps to exactly one point on the Gauss map, which is the direction of its normal.

For two triangles A and B (not necessarily coplanar), let \vec{a} and \vec{b} be position vectors (vectors from the origin to the point) of any points belonging to triangles A and B , respectively. The Minkowski difference P of A and B can be defined $P = \{\vec{p} | \vec{p} = \vec{a} - \vec{b} \text{ s.t. } \vec{a} \in A \text{ and } \vec{b} \in B\}$. If A and B intersect each other, there will be at least one point that is common to both the triangles. The point belonging to the Minkowski difference corresponding to the common point will lie at the

origin. Thus if the two triangles intersect, the Minkowski difference must contain the origin and vice versa: if the Minkowski difference contains the origin, then there must be a point that is common to both the triangles and thus the two triangles must intersect. Furthermore, if the two triangles do not intersect, the minimum distance between them can be given by $\min(|\vec{a} - \vec{b}|)$, where $\vec{a} \in A$, $\vec{b} \in B$. But this is nothing but $\min(|\vec{p}|)$ where $\vec{p} \in P$. Thus the minimum distance between two non-intersecting triangles is given by the minimum distance between the origin and their Minkowski difference.

The Gauss map $G(A)$ of triangle A consists of three great circles (one for each edge) that intersect at two antipodal points. The common points denote the direction normal to the triangle. Likewise we can find the Gauss map $G(B)$ of triangle B . The superposition of these two maps covers all the pairwise features of A and B , respectively, which give rise to the boundary features of the Minkowski difference [16,21]. Thus the Gauss map $G(P)$ of their Minkowski difference P is the superposition of $G(A)$ and $G(B)$. The superposition consists of six great circles, three for each triangle. Assuming that the great circles are in general position, each great circle of triangle A will intersect each great circle of triangle B at two antipodal points, for a total of 9 interacting pairs. Each pair yields two antipodal intersection points, for a total of 18 intersection points. Taken together with the two antipodal points from both $G(A)$ and $G(B)$, in all there are 22 intersection points in the superposition. These intersection points represent directions that are either (1) normal to the plane of one of the triangles or (2) normal to a pair of edges such that one edge is from triangle A and the other is from triangle B . Points of the first type arise from the intersection of three great circles from the same triangle. Points of the second type arise from the intersection of two circles from different triangles. Thus, the planes normal to these directions are either parallel to the plane of a triangle or parallel to a pair of edges.

These planes correspond to the faces of the Minkowski difference of triangles A and B . Faces of the Minkowski difference are formed in two ways. The first case is when triangle A is translated by subtracting a vertex of triangle B from it (or vice versa). This face will be parallel to triangle A , (respectively B) and its Gauss map is the same as $G(A)$, (respectively $G(B)$), yielding the points obtained in case (1) above. The second type of face is formed when an edge of triangle A is translated by subtracting every point on an edge of triangle B (or vice-versa). The face thus formed will be parallel to both the edges and its Gauss map will be the antipodal points obtained by the intersection of the great circles defined by the edges, yielding the points obtained in case (2) above. Thus the intersection points in the superposition of $G(A)$ and $G(B)$ also represent the faces of the Minkowski difference P . Since the points come in antipodal pairs, each face of P has a face parallel to it. Thus P is formed by the intersection of 11 slabs bounded by parallel planes (or fewer slabs if the great circles were not in general position).

Recall that the Minkowski difference P contains the origin if and only if A and B intersect each other and that the minimum

distance between two disjoint convex polytopes is equal to the minimum distance of P from the origin to their Minkowski difference. If the two polytopes penetrate each other, then the depth of penetration is equal to the minimum distance between the origin and the boundary of the Minkowski difference [28]. In the case of our input polytopes A and B , however, they do not penetrate by definition because they are from a non-self-intersecting mesh. Thus if A and B touch each other at the boundary, then the origin will lie on the boundary of P since the penetration depth, as well as the minimum distance between them, is zero.

Thus the question of whether two triangles intersect each other reduces to checking whether the origin lies inside the slabs that form P . Each slab, which is bounded by two parallel planes, defines an axis normal to it. The axial projection of a point p onto an axis is the point of intersection of the axis and the line perpendicular to the axis that passes through the point p . A given slab contains the origin if and only if the axial projections of the triangles onto a normal defined by that slab overlap. In addition, the distance from the origin to the slab equals the length of the gap/overlap between the images of A and B under axial projection onto the axis [21] (Fig. A1).

If the origin lies on the boundary of P , the boundary of at least one slab will pass through the origin. Thus there would be at least one separating axis where the projections of the interiors of the two triangles onto that axis just touch each other, i.e. both the gap and the overlap between them is zero. If the two triangles are completely disjoint, then there will be at least one separating axis where the projections are disjoint.

If the projections are disjoint, then a plane normal to the separating axis that is placed such that it intersects the axis at a point lying inside the gap between the two projections is a separating plane. If the projections just touch each other, then a plane that intersects the axis at the point where both the axial projections meet will separate the interiors of the two triangles and pass through the points on the boundary that are common to both the triangles.

For the case of completely disjoint triangles, the nine vectors from the vertices of triangle A to the vertices of triangle B cross from the open half-space A induced by the separating plane that contains A to the open half-space that contains B , as

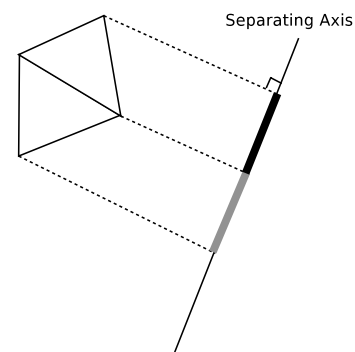


Fig. A1. The projections of two triangles sharing an edge, projected axially on a separating axis, will just touch each other.

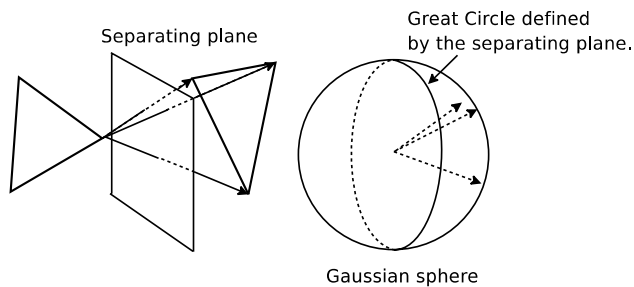


Fig. A2. Vectors joining vertices of one triangle to the vertices of the other triangle cross the separating plane. These vectors, when mapped onto the Gaussian sphere, lie in one of the two hemispheres defined by the separating plane. Only three of the nine vectors are shown to avoid clutter.

shown in Fig. A2. The separating plane defines a great circle on the Gaussian sphere ('equatorial' relative to the 'pole' to which its normal maps) that divides it into two hemispheres. The nine vectors will all map to points on the Gaussian sphere in the open hemisphere corresponding to the half-space containing B . In the case where only the triangle interiors are disjoint, the vectors will cross from the closed half-space containing A to the closed half-space containing B . Thus the vectors will either map to points in the open hemisphere containing B , points on the great circle defining the hemisphere (for vectors defined by non-coincident points both on the separating plane), or to null vectors (for vectors defined by coincident points) that are dropped from further calculations. Thus all the vectors map to points that are in one closed hemisphere of the Gaussian sphere. \square

References

- [1] Ahn H-K, de Berg M, Bose P, Cheng S-W, Halperin D, Matousek J. Separating an object from its cast. *Comput Appl Biosci* 2002;34:547–59.
- [2] Balasubramaniam M, Laxmiprasad P, Sarma S, Shaikh Z. Generating 5-axis NC roughing paths directly from a tessellated representation. *Computer-Aided Des* 2000;32(4):261–77.
- [3] cgal.org. Computational geometry algorithms library; 2004. Available from: <http://www.cgal.org>.
- [4] Chen L-L, Chou S-Y, Woo TC. Parting directions for mould and die design. *Computer-Aided Des* 1993;25(12):762–8.
- [5] Chen L-L, Chou S-Y. Partial visibility for selecting a parting direction in mold and die design. *J Manuf Syst* 1995;14(5):319–30.
- [6] Chen YH. Determining parting direction based on minimum bounding box and fuzzy logics. *Int J Mach Tools Manuf* 1997;37(9):1189–99.
- [7] Chen Y, Rosen DW. A reverse glue approach to automated construction of multi-piece molds. *J Comput Inf Sci Eng* 2003;3(3):219–30.
- [8] Chen Y. Computer-aided design for rapid tooling: methods for mold design and design-for-manufacture. PhD Thesis. Georgia Institute of Technology; 2001.
- [9] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry: algorithms and applications. 2nd ed. Berlin: Springer; 2000.
- [10] Dhaliwal S, Gupta S, Huang J, Kumar M. A feature based approach to automated design of multi-piece sacrificial molds. *ASME J Comput Inf Sci Eng* 2001;1(3):225–34.
- [11] Dhaliwal S, Gupta S, Huang J, Priyadarshi A. Algorithms for computing global accessibility cones. *J Comput Inf Sci Eng* 2003;3(3):200–9.
- [12] Elber G, Chen X, Cohen E. Mold accessibility via gauss map analysis. *J Comput Inf Sci Eng* 2005;5(2):79–85.
- [13] Elber G, Sayegh R, Barequet G, Martin RR. Two dimensional visibility atlases for continuous curves. In: Proceedings of international conference on shape modeling and application; 2005. p. 301–8.
- [14] Everitt C, Rege A, Cebenoyan C. Interactive geometric and scientific computation using graphics hardware. *ACM SIGGRAPH 2003 Course #11 Notes*; 2003.
- [15] Fernando R. GPU gems programming techniques, tips and tricks for real time graphics hardware. Reading, MA: Addison-Wesley; 2004.
- [16] Fogel E, Halperin D. Video: exact minkowski sums of convex polyhedra. In: Proceedings of 21st ACM symposium on computational geometry; 2005, pp. 382–83.
- [17] Fu MW, Fuh JYH, Nee AYC. Generation of optimal parting direction based on undercut features in injection molded parts. *IIE Trans* 1999;31:947–55.
- [18] Fu MW, Fuh JYH, Nee AYC. Undercut feature recognition in an injection mould design system. *Computer-Aided Des* 1999;31(12):777–90.
- [19] Ganter MA, Skoglund PA. Feature extraction for casting core development. In: 17th design automation conference presented at the 1991 ASME design technical conferences. American Society of Mechanical Engineers, Miami, FL; 1991, pp. 93–100.
- [20] Gan JG, Woo TC, Tang K. Spherical maps: their construction, properties and approximation. *J Mech Des* 1994;116(2):357–63.
- [21] Gottschalk S. Collision queries using oriented bounding boxes. PhD Thesis. University of North Carolina, Chapel Hill, Department of Computer Science; 1998. Available from: citeseer.ist.psu.edu/gottschalk00collision.html.
- [22] Hui KC, Tan ST. Mould design with sweep operations—a heuristic search approach. *Computer-Aided Des* 1992;24(2):81–91.
- [23] Inui M, Kakio R. Fast visualization of NC milling result using graphics acceleration hardware. In: IEEE international conference on robotics and automation. IEEE, San Francisco, CA; 2000. p. 3089–94.
- [24] Inui M. Fast inverse offset computation using polygon rendering hardware. *Computer Aided Des* 2003;35(2):191–201.
- [25] Khardekar R, Burton G, McMains S. Finding feasible mold parting directions using graphics hardware. In: Proceedings of the 2005 ACM symposium on solid and physical modeling, Cambridge, MA; 2005. p. 233–43.
- [26] Khardekar R, McMains S. Finding mold removal directions using graphics hardware. In: ACM workshop on general purpose computing on graphics processors; 2004, pp. C-19, (abstract). Available from: www.cs.unc.edu/Events/Conferences/GP2.
- [27] Kilgard M. SIGGRAPH 2002 course notes. vol. 31. ACM SIGGRAPH, San Antonio, TX, Ch. Interactive geometric computations using graphics hardware; 2002.
- [28] Kim Y, Otaduy M, Lin M, Manocha D. Fast penetration depth computation for physically-based animation; 2002. Available from: citeseer.ist.psu.edu/article/kim02fast.html.
- [29] König AH, Groller E. Real time simulation and visualization of NC milling processes for inhomogeneous materials on low-end graphics hardware. In: Computer Graphics International IEEE; 1998. p. 338–49.
- [30] Kwong K. Computer-aided parting line and parting surface generation in mould design. PhD Thesis. The University of Hong Kong, Hong Kong; 1992.
- [31] McGuire M, Hughes JF. Hardware-determined feature edges. In: Proceedings of the third international symposium on non-photorealistic animation and rendering. ACM Press; 2004. p. 35–147.
- [32] McMains S, Chen X. Determining moldability and parting directions for polygons with curved edges. In: International mechanical engineering congress and exposition. ASME, Anaheim, CA; 2004. pp. IMECE2004-62227.
- [33] O'Rourke J. Computational geometry in C. 2nd ed. Berlin: Springer; 1998.
- [34] Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, et al. A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, State of the art reports; 2005. p. 21–51.

- [35] Pop M, Duncan C, Barequet G, Goodrich M, Huang W, Kumar S. Efficient perspective-accurate silhouette computation and applications. In: Proceedings of the seventeenth annual symposium on computational geometry. ACM Press; 2001. p. 60–8.
- [36] Priyadarshi A, Gupta S. Geometric algorithms for automated design of multi-piece permanent molds. *Computer Aided Des* 2004;36(3): 241–60.
- [37] Rappaport D, Rosenbloom A. Moldable and castable polygons. *Comput Geom: Theory Appl* 1994;4:219–33.
- [38] Ravi B, Srinivasan MN. Decision criteria for computer-aided parting surface design. *Computer-Aided Des* 1990;22:11–18.
- [39] Reeves W, Salesin D, Cook R. Rendering antialiased shadows with depth maps. In: Proceedings of the 14th annual conference on computer graphics and interactive techniques SIGGRAPH 87; 1987. p. 283–91.
- [40] Requicha AAG. Representations for rigid solids: theory, methods and systems. New York: ACM Computing Surveys; 1980. p. 437–464.
- [41] Saito T, Takahashi T. NC machining with G-buffer method. *SIGGRAPH* 91 1991;25(4):207–16.
- [42] Shreiner D, Woo M, Neider J, Davis T. *OpenGL programming guide: the official guide to learning OpenGL Version 1.4*. Addison-Wesley, Ch. Blending, Antialiasing, Fog and Polygon Offset; 2004. p. 268–70.
- [43] Spitz S, Spyridi A, Requicha A. Accessibility analysis for planning of dimensional inspection with coordinate measuring machines. *IEEE Trans Robotom Autom* 1999;714–27.
- [44] Stampfl J, Liu H-C, Nam SW, Sakamoto K, Tsuru H, Kang S, et al. Rapid prototyping and manufacturing by gelcasting of metallic and ceramic slurries. *Mater Sci Eng* 2002;334(1–2):187–92.
- [45] Williams L. Casting curved shadows on curved surfaces. In: Proceedings of the fifth annual conference on computer graphics and interactive techniques SIGGRAPH 87; 1978. p. 270–74.
- [46] Wong T, Tan ST, Sze WS. Parting line formation by slicing a 3D CAD model. *Eng Comput* 1998;14(4):330–43.
- [47] Woo TC. Visibility maps and spherical algorithms. *Computer-Aided Des* 1994;26(1).
- [48] Wuerger D, Gadh R. Virtual prototyping of die design part one: theory and formulation. *Concurrent Eng: Res Appl* 1997;5(4):307–15.
- [49] Wuerger D, Gadh R. Virtual prototyping of die design part two: algorithmic, computational, and practical considerations. *Concurrent Eng: Res Appl* 1997;5(4):317–26.
- [50] Ye XG, Fuh JYH, Lee KS. A hybrid method for recognition of undercut features from moulded parts. *Computer-Aided Des* 2001;33:1023–34.
- [51] Yin Z, Ding H, Xiong Y. Virtual prototyping of mold design: geometric mouldability analysis for near-net-shape manufactured parts by feature recognition and geometric reasoning. *Comput-Aided Des* 2001;33: 137–54.

Rahul Khardekar is pursuing his doctoral studies in Mechanical Engineering at the University of California, Berkeley. He received his B.E. (Mechanical) from the Government College of Engineering at the University of Pune (India), and his M.S.E. (Mechanical) from the University of Texas at Austin. His research interests include computer-aided design, computer graphics, scientific visualization, and general purpose computation on graphics hardware.

Greg Burton has a B.S. in mechanical engineering from UC San Diego and a M.S. in mechanical engineering specializing in mechatronics and controls at UC Berkeley. His research interests have alternatively focused between scientific visualization, solid modeling, and robotics. He is currently working as a research engineer at Lockheed Martin Space Systems Company's Advanced Technology Center in Palo Alto researching robotic simulation, and sensor simulation.

Prof. Sara McMains Dr. McMains is an Assistant Professor in the Department of Mechanical Engineering, University of California, Berkeley. Her research interests include geometric algorithms, computer aided design & manufacturing, computer aided process planning, layered manufacturing geometric and solid modeling, computer graphics and visualization virtual prototyping, and virtual reality. She received her A.B. from Harvard University in Computer Science, and her M.S. and Ph.D. from UC Berkeley in Computer Science with a minor in Mechanical Engineering. She is the recipient of Best Paper Awards from Usenix (1995) and ASME DETC (2000), and the NSF CAREER Award (2006).