to appear in the 2005 ACM Symposium on Solid and Physical Modeling (SPM)

Finding Feasible Mold Parting Directions Using Graphics Hardware

Rahul Khardekar Greg Burton Sara McMains

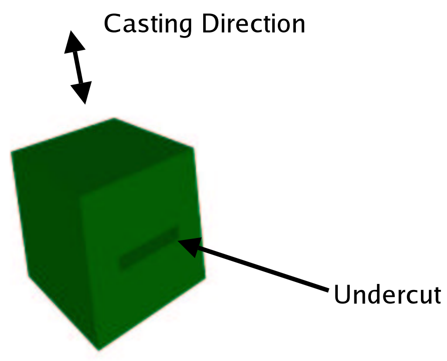
Mechanical Engineering Department
University of California, Berkeley [*]

We present new programmable graphics hardware accelerated algorithms to test the castability of geometric parts and assist with part redesign. These algorithms efficiently identify and graphically display undercuts and 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 castability using accessibility analysis and Gauss maps. The efficiency of these algorithms lies in that they identify groups of candidate directions such that if any one direction in the group is not castable, none are, or if any one is castable, all are. We examine trade-offs between the algorithms' speed, accuracy, and whether they guarantee that a castable direction will be found for a part if one exists.

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 "casting direction") to remove the part. To be castable in a given direction, it must be free from undercut features which would make it impossible to define mold halves that could be separated from the part when translated along the casting directions without colliding with the part (see Figure 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, mold halves are translated only, always in opposite directions.

We call an object "castable" in a potential mold removal direction if it has no undercuts relative to that direction; the direction is called a castable direction for that object. Formally, an object is castable in a direction $\vec{d}$ if the complement of the object can be split in 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 [Ahn et al. 2002]. Note that if the object is castable in a direction $\vec{d}$, then it is also castable in $-\vec{d}$. For a convex part all directions are castable directions. For other part geometries, there may be *no* castable 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



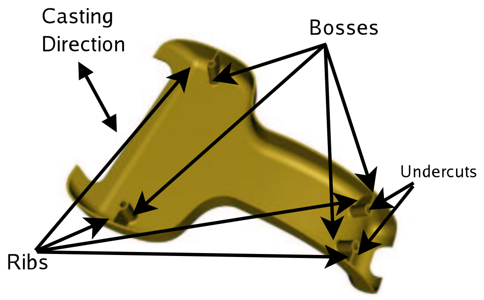


Figure 1: Parts which are not castable in the vertical direction because of undercut features (a) Cube with an undercut (b) Plastic casing with undercuts on lower cylindrical bosses

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.

In the design process, the casting direction should be determined before detail design features like the bosses and ribs in Figure 1-b are added. There are numerous occasions when a part needs to be modified after the casting direction is chosen. Thus, there is a need for a tool which will warn the designer as soon as a change is made which makes the part non-castable in the chosen direction. Earlier in the design process, during the conceptual design phase, feedback about whether *any* castable directions exist for the proposed geometry is helpful. During this stage, identifying all possible castable directions is useful because a designer can then choose the best possible direction in terms of manufacturing cost. In this paper, we describe efficient graphics-hardware accelerated algorithms to solve the above problems for tessellated input geometry.

[*]{rahul | mcmains}@me.berkeley.edu, greg.burton@gmail.com

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 [Lin and Manocha 2003]. 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 [Saito and Takahashi 1991; Konig and Groller 1998; Balasubramaniam et al. 2000; Spitz et al. 1999; Inui and Kakio 2000; Inui 2003], but today's programmable hardware can speed up more complex calculations [Fernando 2004]. The preliminary results we described in [Khardekar and McMains 2004] were the first application of the programmable capabilities of GPUs to castability that we are aware of.

A programmable graphics card allows general purpose computing in two stages of the graphics rendering pipeline. A vertex shader 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 shader executes a user defined fragment program at every pixel, taking the interpolated vertex and texture 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 castability and finding castable directions. Many researchers who have looked at the problem of finding a casting direction for a two-part mold for a given geometry only look at a limited number of potential directions. [Wong et al. 1998] and [Ravi and Srinivasan 1990] only consider parting directions along the three principle axes. [Chen 1997] looks only at the axes of a minimum bounding box. [Hui and Tan 1992] use a heuristic search approach which, even though not exhaustive, shows significant performance hits on more complex parts with curved surfaces.

[Dhaliwal et al. 2001] 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 [Chen 2001; Chen and Rosen 2003] and shape deposition modeling for sacrificial molds [Stampfl et al. 2002].

Chen et al. introduce the term visibility map to the demoldability literature in a paper that shows how to minimize the number of cores in parts that can't necessarily be made in a two-part mold [Chen et al. 1993]. They find potential undercuts by performing a regularized Boolean subtraction [Requicha 1980] 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 [Woo 1994]. 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 doesn't admit a two-part mold and describe how potential undercuts that can't be handled by a single core can be subdivided to show a designer what would need to be changed to make a design moldable [Chen and Chou 1995].

A number of papers look to graph-based feature recognition methods to find potential undercuts [Ganter and Skoglund 1991; Fu et al. 1999a; Fu et al. 1999b; Yin et al. 2001]. Unfortunately graph-based methods break down for interacting features, a shortcoming that [Ye et al. 2001] address by using a hybrid approach that doesn't rely exclusively on graph matching. [Wuerger and Gadh 1997a] present an incomplete algorithm based on convex hull differences, similar to the [Chen et al. 1993] approach, to find a parting direction for a two-part mold. Their more significant contribution is in their companion paper [Wuerger and Gadh 1997b], 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 castability can be found in the computational geometry literature. [Rappaport and Rosenbloom 1994] present an $O(n \log n)$ time algorithm (unimplemented) for the 2D case of finding if a polygon is 2-castable. Ahn et al. combines strong theory with a partial implementation [Ahn et al. 2002]. They show that a definitive answer to whether a polyhedron is castable 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, [McMains and Chen 2004] analyze 2D curved spline input, and for 3D [Elber et al. 2004] describe an impressive new algorithm combining strong theory and an implementation, but it is limited to C^3 continuous NURBS surfaces only.

3 Checking a direction for castability

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

3.1 Castability checking

For simplicity, we first describe the castability checking algorithm assuming a vertical casting direction. We define a part facet as an "up-facet" with respect to a given direction $\vec{d}$ if the angle between the facet's outward facing surface normal and $\vec{d}$ is less than $90°$. For a vertical casting 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. 2002] proved that a given part geometry is vertically castable 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 castable and hence 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 casting direction, the part is castable 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 [Ahn et al. 2002; Kwong 1992]. 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 vertically castable. Figure 2 shows a two dimensional example of this process for a polygon (assume the edges are 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 castable in the vertical direction.

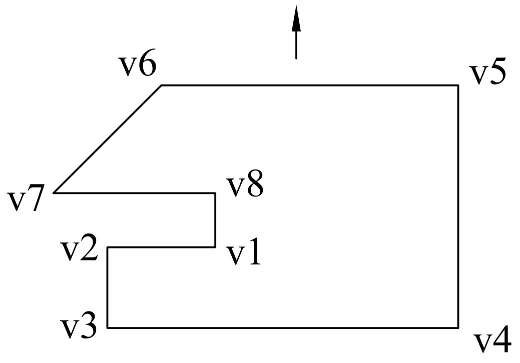


Figure 2: Two dimensional example of castability test

Thus determining whether or not a part is castable 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 casting 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 -d
   c. eye point offset +d from part bounds
   d. screen parameters 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 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 castable in direction $\vec{d}$. On recent graphics cards, we can efficiently check if any pixels were rendered in this pass by using the graphics card’s occlusion query functionality, rather than reading back the entire frame-buffer, making using the frame-buffer more efficient than using the stencil-buffer, for example. Figure 3 shows an example of the the frame-buffer after step 3 and again after step 5 when the algorithm is run on the part in Figure 1-b with a vertical casting direction.

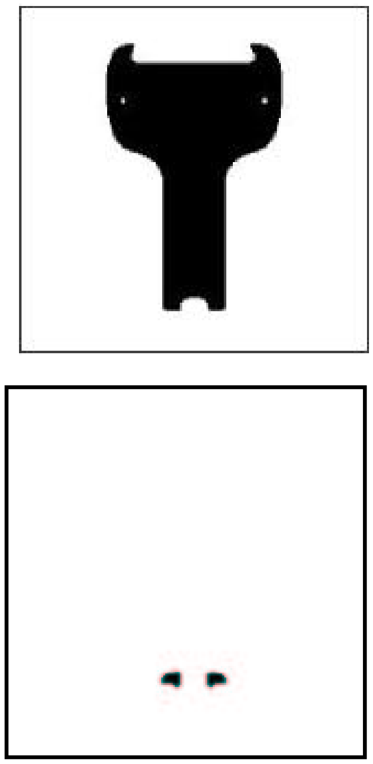


Figure 3: Screen shots of the frame-buffer showing pixels rendered after the first (top) and second pass (bottom) of the algorithm

Our implementation of this algorithm, running on a QuadroFX 3000 GPU, was able to test the castability of parts with over 20,000 facets in less than one millisecond per direction tested using a 256x256 frame-buffer (Figure 4). Our direction testing rate ranged from 18,200 to 1,040 directions per second, on parts with 40 to 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 castability (by 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 [Pop et al. 2001], but the $O(n \log n)$ plane sweep intersection testing still dominates the theoretical complexity. (GPU algorithms for silhouette extraction designed for rendering, such as [McGuire and Hughes 2004], do not seem appropriate for this purpose 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.)

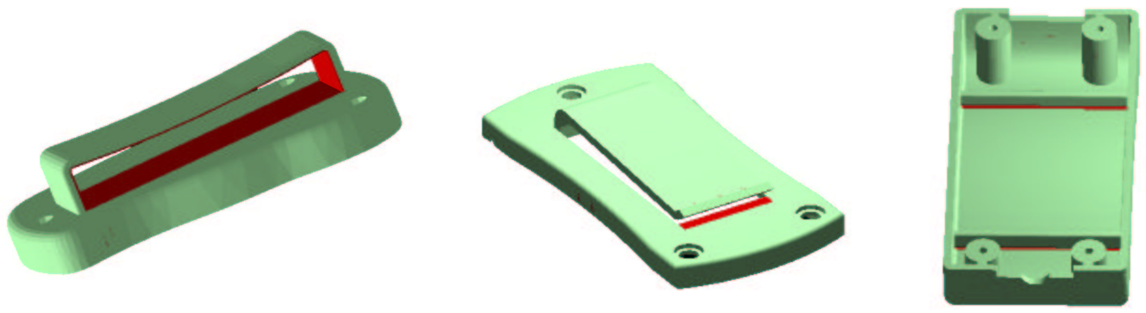


Figure 5: Undercuts in prototype part designs found and highlighted by our software (vertical removal direction)

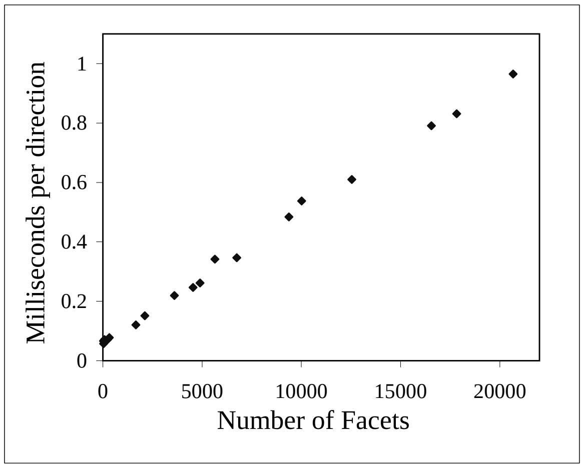


Figure 4: Running times (QuadroFX 3000 GPU, AMD Athlon 1.8 GHz CPU)

3.2 Highlighting non-castable features

Once it is determined that an object is not castable for a given mold-removal direction, we would like to highlight the portions of the surface that are not castable so that the designer can make the necessary corrections. Up-facets which are rendered in the second pass of the two pass algorithm along with down-facets which are invisible if the object is viewed from the opposite direction are the non-castable facets, 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 non-castable surfaces will be exactly those in shadow. For the 2D example in Figure 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 which store depth values at each pixel location, allowing a second depth test for each pixel as described in [Everitt et al. 2003].

As a preprocessing step before we can display the part with its non-castable features highlighted, we perform the following procedure twice, once from the positive casting direction and once from the negative casting direction. First the scene is projected orthographically with the camera placed above the part, aligned along the positive casting 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 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 casting 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. Thus after interpolation during rasterization, the first two components of a pixel texture coordinate give the location of that pixel in the associated depth texture and the third component gives the depth of that pixel from the positive (alternately, negative) casting 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 casting directions; we highlight it in red to indicate that it is on a non-castable undercut. The three parts shown in Figure 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 a slight taper or “draft” should be applied to all vertical walls to facilitate mold removal. The angle the modified faces then make with the vertical, typically one to three degrees, 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 casting 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 casting direction and the facet normal, and compare it to the stored sine of the threshold draft angle, modifying the display color of the triangles with angle less than the desired value.

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, for the three vertices 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 casting direction as constant data, and then take a dot product between the normal of the facet (stored with the vertex) and the casting direction, thus calculating the cosine of the angle between the facet normal and the casting direction, equivalent to the sine of the draft angle (Figure 6). 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 (Figure 7). 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 that value.

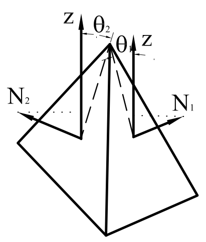


Figure 6: The dot product of the normal and the casting direction (here, +z) is the sine of the draft angle for a face

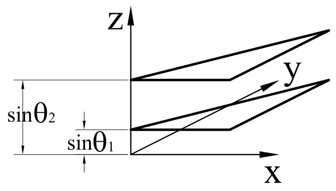


Figure 7: Dummy triangles rendered for every facet to find the minimum draft angle

Figure 8 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 hundred facets, the GPU algorithm is a clear win. Its incremental cost for additional facets is negligible, while the software algorithm time continues to grow linearly with the number of facets.

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

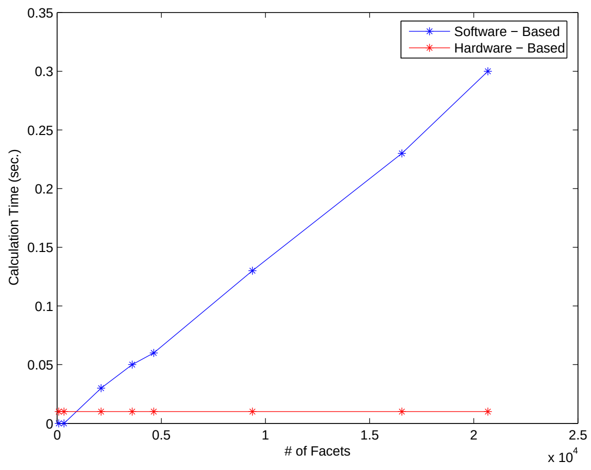


Figure 8: Performance of draft analysis algorithm

4 Finding castable directions

In this section we describe two algorithms we have developed and implemented to choose candidate directions to test for castability. 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 castable, none are, or if any one is castable, 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 which is perpendicular to the normal vector of the plane. This great circle divides the sphere in 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 on the GPU. It is inspired by the theoretical algorithm of Ahn et al., who prove that all combinatorially distinct casting directions correspond to 0-, 1-, or 2-cells in an arrangement of great circles on a Gaussian sphere [Ahn et al. 2002]. 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).

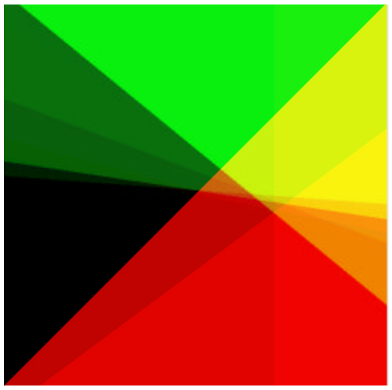


Figure 9: Arrangement of lines in a quadtree leaf node

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 per color channel and we allocate one bit per line; for the latest graphics cards with 32 bits per color channel we would use 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). Figure 9 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 1,024 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 before we've reduced the number of lines, as will always happen if more than 32 lines go through the same point, we stop subdividing and just pick 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 castability.

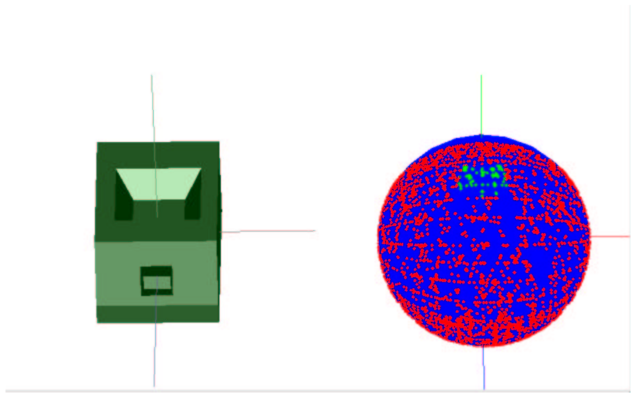


Figure 11: Sample part with castable directions shown by the (lighter) green dots on the Gaussian sphere. Non-castable directions are shown by (darker) red dots.

Figure 11 shows a part, alongside the directions tested displayed on the Gaussian sphere. The (lighter) green dots on the sphere represent the castable directions found and the (darker) red dots represent the directions which were checked and found non-castable. Table 10 shows timing data for additional parts we tested.

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 castable directions. Also we found that the speed of frame-buffer read-back was too slow to be useful for practical applications with parts with a large number of facets.

4.2 Convex hull intersection algorithm

Our second, more accurate algorithm 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 [Gan et al. 1994]. 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 [Dhaliwal et al. 2003]. 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 non-castable in a given direction only if any two up-facets overlap each other when projected orthographically on a plane normal to that direction. Thus only pairs of facets which can potentially become up-facets simultaneously, with their projections also overlapping (non-disjoint interiors) each other, could affect the castability of the object in *any* direction. We call such pairs of facets "potentially interacting" facets. 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-castable 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 many such pairs a priori.

Lemma: 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 Figure 12). Call the great circles on the Gaussian sphere perpendicular to the normal vectors of A and B C_A and C_B respectively. Each great circle divides the sphere in 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.

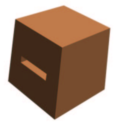
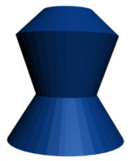
				
No. facets	28	40	80	328
No. great circles	128	258	863	1832
No. directions tested	6,085	17,418	283,904	476,646
No. castable directions found	1	2	34,758	120,299
Time (secs)	2.58	6.89	78	200

Figure 10: Performance data for quadtree algorithm

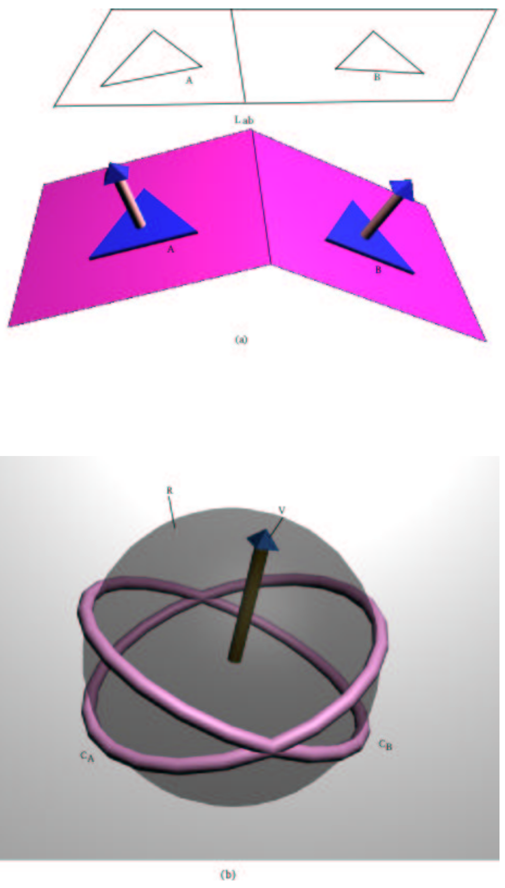


Figure 12: (a) A pair of noninteracting triangles (b) Gaussian sphere with great circles corresponding to the pair of triangles

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 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 which 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.

□

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 castable in any direction.

Now imagine moving over the surface of the Gaussian sphere, considering different casting directions. The only event that changes the castability of our current direction is when a pair of potentially interacting facets start or stop occluding each other. Thus the castability 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 castability does not change in the interior of any region. Furthermore, if the object is castable in the directions interior to a region, then it is also castable 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 castable in directions corresponding to the interior points of the arc it is also castable in the directions corresponding to the arc boundaries, the two vertices of the arc segment. On the other hand, note that the object may be castable in directions along an arc separating two regions that are not themselves castable, and it may be castable in the directions corresponding to the vertices of an arc whose other (interior) points don't correspond to castable directions. Thus to check whether there exists any castable direction for the object, it suffices to test the castability at the vertices of the connected regions. These vertices may be the convex hull vertices or they may be new vertices introduced by the intersection of the original convex hull arcs.

Unlike the previous algorithm, this algorithm is theoretically guaranteed to include a castable direction in the set of candidate directions if any castable direction exists. Of course, in practice, there will be roundoff 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 castability 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 Figure 15, 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 every vertex in the other facet [Dhaliwal et al. 2003]. 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. This follows from the fact that for any two triangles with disjoint interiors, there exists a separating plane such that each triangle is in opposite closed half-spaces defined by this plane. The vectors all originate from the same side of the separating plane and cross to the other side, so all the vectors map to points on the same hemisphere defined by the great circle corresponding to the separating plane normal. The existence of a separating plane for two disjoint convex facets, and the fact that it can be found efficiently because a separating plane exists parallel to one of the facets or parallel to an edge of each facet, follows from the separating plane theorem [Gottschalk 1998]. On the 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 [Gan et al. 1994].

We tried three different convex hull calculation approaches. The first was the gift-wrapping convex hull algorithm as described in [de Berg et al. 2000]. 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) [cgal.org 2004], Bykat's non-recursive version of quickhull as well as Akl and Toussaint's 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 near-collinear points.

We had the best luck when we implemented Graham's scan algorithm as described in [O'Rourke 1998] 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 roundoff errors calculating the angles will only cause small quantitative errors in the results, not mutually impossible answers to sidedness queries.

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 hemicube, 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. Figure 13 shows this process for a pair of triangles.

We first check the vertices of these lines to see if they represent castable directions. If none of the vertices are castable, then we find the intersections of the lines and check those intersections for castability (Figure 14). If we are only interested in finding whether the object is castable, we can stop after we find a castable 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

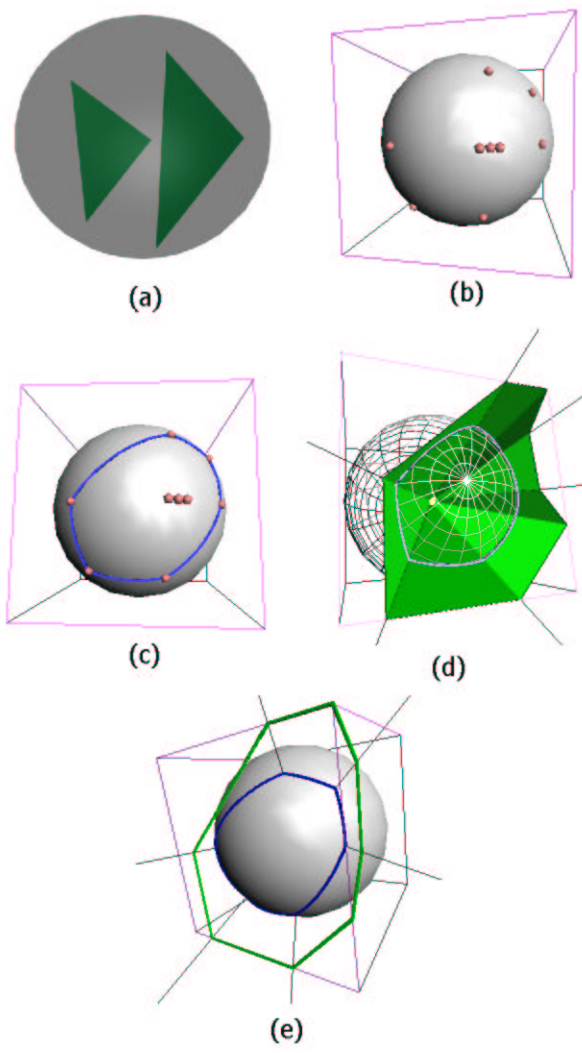


Figure 13: (a) A pair of triangles (b) Points on the Gaussian sphere obtained from the pair of triangles (c) Spherical convex hull obtained on the sphere (d) Projecting the convex hull on a circumscribing cube (e) The convex hull obtained on the faces of the cube after splitting.

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

Details of running times to find a castable direction for different parts are given in Figure 15. The convex hull calculations for the more complex parts take from seconds to minutes; castability 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 castable direction, which our algorithm only found after about fifteen minutes of processing.

5 Future work

We will continue working to optimize our original proof-of-concept implementation of the convex hull intersection algorithm. Currently we test all the vertices we generate, even those that lie in

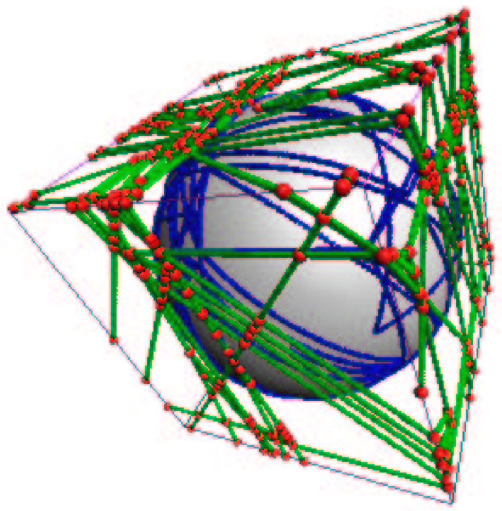


Figure 14: Points of intersections of lines on the hemi-cube faces

the interior of an inaccessibility region of another pair of facets. Because the inaccessibility regions are regions of non-castability, such vertices will always be non-castable. We will investigate 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 castability of such vertices. A promising approach could be to find the (non-regularized) union of the interiors of the non-castable regions and only test vertices of the 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 castability 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, we suspect the problem of finding a mold design that is guaranteed to minimize the number of mold pieces is NP-hard. But we believe that GPU-based algorithms can be used within heuristic approaches to facilitate the design of parts to be manufactured in multi-piece molds as well.

6 Acknowledgments

The graphics card used in this work was donated by NVIDIA. The authors were supported in part by UC MICRO. We would also like to thank the reviewers for comments that improved the exposition.

References

AHN, H.-K., DE BERG, M., BOSE, P., CHENG, S.-W., HALPERIN, D., MATOUSEK, J., AND SCHWARZKOPF, O. 2002. Separating an object from its cast. *Computer-Aided Design 34*, 547–59.

BALASUBRAMANIAM, M., LAXMIPRASAD, P., SARMA, S., AND SHAIKH, Z. 2000. Generating 5-axis NC roughing paths directly from a tessellated representation. *Computer-Aided Design 32*, 4 (April), 261–77.

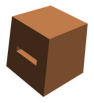
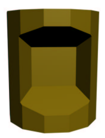
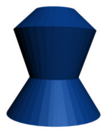
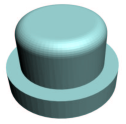
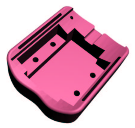
					
No. facets	28	80	328	1994	4634
No. pairs	378	3,160	53,628	1,987,021	10,734,661
No. potentially interacting pairs	140	749	15,228	202,050	3,089,765
Time (secs) C.H. calculations	.01	.04	.78	12.37	184.35
Time (secs) castability testing	<.01	<.01	<.01	.09	582.16
No. directions tested	1	7	26	358	1,470,416

Figure 15: Running time for Convex Hull Intersection Algorithm

CGAL.ORG, 2004. Computational Geometry Algorithms Library. http://www.cgal.org.

CHEN, L.-L., AND CHOU, S.-Y. 1995. Partial Visibility for Selecting a Parting Direction in Mold and Die Design. *Journal of Manufacturing Systems 14*, 5, 319–330.

CHEN, Y., AND ROSEN, D. W. 2003. A reverse glue approach to automated construction of multi-piece molds. *Journal of Computing and Information Science in Engineering 3*, 3, 219–230.

CHEN, L.-L., CHOU, S.-Y., AND WOO, T. C. 1993. Parting directions for mould and die design. *Computer-Aided Design 25*, 12 (December), 762–768.

CHEN, Y. H. 1997. Determining parting direction based on minimum bounding box and fuzzy logics. *Int. J. Mach. Tools Manufact. 37*, 9, 1189–1199.

CHEN, Y. 2001. *Computer-Aided Design for Rapid Tooling: Methods for Mold Design and Design-for-Manufacture*. PhD thesis, Georgia Institute of Technology.

DE BERG, M., VAN KREVELD, M., OVERMARS, M., AND SCHWARZKOPF, O. 2000. *Computational Geometry: Algorithms and Applications*, second ed. Springer, Berlin.

DHALIWAL, S., GUPTA, S., HUANG, J., AND KUMAR, M. 2001. A feature based approach to automated design of multi-piece sacrificial molds. *ASME Journal of Computing and Information Science in Engineering 1*, 3, 225–234.

DHALIWAL, S., GUPTA, S., HUANG, J., AND PRIYADARSHI, A. 2003. Algorithms for computing global accessiblity cones. *Journal of Computing and Information Science in Engineering 3*, 3 (September), 200–209.

ELBER, G., CHEN, X., AND COHEN, E. 2004. Mold Accessibility via Gauss Map Analysis. In *Proceedings of Shape Modeling International*, 263–72.

EVERITT, C., REGE, A., AND CEBENOYAN, C., 2003. Interactive geometric and scientific computation using graphics hardware. ACM SIGGRAPH 2003 Course #11 Notes, July.

FERNANDO, R. 2004. *GPU Gems Programming Techniques, Tips amd Tricks for Real Time Graphics Hardware*. Addison-Wesley.

FU, M. W., FUH, J. Y. H., AND NEE, A. Y. C. 1999. Generation of optimal parting direction based on undercut features in injection molded parts. *IIE Transactions 31*, 947–955.

FU, M. W., FUH, J. Y. H., AND NEE, A. Y. C. 1999. Undercut feature recognition in an injection mould design system. *Computer-Aided Design 31*, 12 (October), 777–790.

GAN, J. G., WOO, T. C., AND TANG, K. 1994. Spherical maps: their construction, properties and approximation. *Journal of Mechanical Design 116*, 2 (June), 357–363.

GANTER, M. A., AND SKOGLUND, P. A. 1991. Feature extraction for casting core development. In *17th Design Automation Conference presented at the 1991 ASME Design Technical Conferences*, American Society of Mechanical Engineers, 93–100.

GOTTSCHALK, S. 1998. *Collision Queries using Oriented Bounding Boxes*. PhD thesis, University of North Carolina, Chapel Hill, Department of Computer Science.

HUI, K. C., AND TAN, S. T. 1992. Mould design with sweep operations - a heuristic search approach. *Computer-Aided Design 24*, 2 (February), 81–91.

INUI, M., AND KAKIO, R. 2000. Fast visualization of NC milling result using graphics acceleration hardware. In *IEEE International Conference on Robotics and Automation*, IEEE, 3089–94.

INUI, M. 2003. Fast inverse offset computation using polygon rendering hardware. *Computer Aided Design 35*, 2 (February), 191–201.

KHARDEKAR, R., AND MCMAINS, S. 2004. Finding mold removal directions using graphics hardware. In *ACM Workshop on General Purpose Computing on Graphics Processors*, C–19. (abstract).

KONIG, A. H., AND GROLLER, E. 1998. Real time simulation and visualization of NC milling processes for inhomogeneous materials on low-end graphics hardware. In *Computer Graphics International*, IEEE, 338–49.

KWONG, K. 1992. *Computer-aided parting line and parting surface generation in mould design*. PhD thesis, The University of Hong Kong, Hong Kong.

LIN, M. C., AND MANOCHA, D. 2003. *SIGGRAPH 2003 Course Notes*, vol. 11. ACM SIGGRAPH, July, ch. Interactive Geometric and Scientific Computations Using Graphics Hardware, 1–6.

MCGUIRE, M., AND HUGHES, J. F. 2004. Hardware-determined feature edges. In *Proceedings of the 3rd international symposium on non-photorealistic animation and rendering*, ACM Press, 35–147.

MCMAINS, S., AND CHEN, X. 2004. Determining Moldability and Parting Directions for Polygons with Curved Edges. In *International Mechanical Engineering Congress and Exposition*, ASME, IMECE2004–62227.

O'ROURKE, J. 1998. *Computational Geometry in C (2nd Edition)*. Springer.

POP, M., DUNCAN, C., BAREQUET, G., GOODRICH, M., HUANG, W., AND KUMAR, S. 2001. Efficient perspective-accurate silhouette computation and applications. In *Proceedings of the seventeenth annual symposium on computational geometry*, ACM Press, 60–68.

RAPPAPORT, D., AND ROSENBLOOM, A. 1994. Moldable and castable polygons. *Computational Geometry: Theory and Applications 4*, 219–233.

RAVI, B., AND SRINIVASAN, M. N. 1990. Decision criteria for computer-aided parting surface design. *Computer-Aided Design 22*, 11–18.

REQUICHA, A. A. G. 1980. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Computing Surveys* (December), 437–464.

SAITO, T., AND TAKAHASHI, T. 1991. NC machining with G-buffer method. *SIGGRAPH 91 25*, 4 (July), 207–16.

SPITZ, S., SPYRIDI, A., AND REQUICHA, A. 1999. Accessibility analysis for planning of dimensional inspection with coordinate measuring machines. *IEEE Transactions on Robotics and Automation*, 714–27.

STAMPFL, J., LIU, H.-C., NAM, S. W., SAKAMOTO, K., TSURU, H., KANG, S., COOPER, A. G., NICKEL, A., AND PRINZ, F. B. 2002. Rapid prototyping and manufacturing by gelcasting of metallic and ceramic slurries. *Materials Science & Engineering. 334*, 1-2 (Sep), 187–192.

WONG, T., TAN, S. T., AND SZE, W. S. 1998. Parting line formation by slicing a 3D CAD model. *Engineering with Computers 14*, 4, 330–343.

WOO, T. C. 1994. Visibility maps and spherical algorithms. *Computer-Aided Design 26*, 1 (January).

WUERGER, D., AND GADH, R. 1997. Virtual Prototyping of Die Design Part One: Theory and Formulation. *Concurrent Engineering : Research and Applications 5*, 4 (December), 307–315.

WUERGER, D., AND GADH, R. 1997. Virtual Prototyping of Die Design Part Two: Algorithmic, Computational, and Practical Considerations. *Concurrent Engineering : Research and Applications 5*, 4 (December), 317–326.

YE, X. G., FUH, J. Y. H., AND LEE, K. S. 2001. A hybrid method for recognition of undercut features from moulded parts. *Computer-Aided Design 33*, 1023–1034.

YIN, Z., DING, H., AND XIONG, Y. 2001. Virtual prototyping of mold design: geometric mouldability analysis for near-net-shape manufactured parts by feature recognition and geometric reasoning. *Computer-Aided Design 33*, 137–154.