Fast 3D Geometric Proximity Queries between Rigid and Deformable Models Using Graphics Hardware Acceleration

Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, Dinesh Manocha
University of North Carolina at Chapel Hill
{hoff,andrewz,lin,dm}@cs.unc.edu

Abstract

We present an approach for computing generalized proximity information between arbitrary polygonal models using graphics hardware acceleration. Our algorithm combines object-space localization, multi-pass rendering techniques, and accelerated distance field computation to perform complex proximity queries at interactive rates. It is applicable to any closed, possibly non-convex, polygonal object and requires no precomputation, making it suitable for both rigid and dynamically deformable geometry of relatively high complexity. The proximity queries include, not only collision detection, but also the computation of intersections, minimum separation distance, closest points, penetration depth and direction, and contact points and normals. The load is balanced between CPU and graphics subsystems through a hybrid geometry and image-based approach. Geometric object-space techniques coarsely localize potential interactions between two objects, and image-space techniques accelerated with graphics hardware provide the low-level proximity information. We have implemented our system using the OpenGL graphics library and have tested it on various hardware configurations with a wide range of object complexities and contact scenarios. In all cases, interactive frame rates are achieved. In addition, our algorithm's performance is heavily based on the graphics hardware computational power growth curve which has exceeded the expectations of Moore's Law for general CPU power growth.

1. Introduction

Many applications of computer graphics or computer simulated environments require spatial or proximity relationships between objects. In particular, dynamic simulation, haptic rendering, surgical simulation, robot motion planning, virtual prototyping, and computer games often need to perform different proximity queries at interactive rates. The set of queries include collision detection, intersection, closest point computation, minimum separation distance, penetration depth, and contact points and normals. Algorithms to perform different queries have been well studied in computer graphics, virtual environments, robotics and computational geometry. Most of the current approaches involve considerable pre-processing and therefore are not fast enough for deformable models. Furthermore, no good algorithms are known for penetration depth computation between general, non-convex models.

We present a novel approach to perform all the proximity queries between rigid and deformable models using graphics hardware acceleration. Our algorithm localizes potential interactions using object-space techniques, point-samples the region, and then uses polygon rasterization hardware to compute object intersections, closest points, and the distance field and its gradients.

The main features of our approach include a unified framework for all proximity queries, applicability to non-convex polygonal models, computational efficiency allowing interactive queries on current PCs, robustness in terms of not dealing with any special-cases or degeneracies, and portability across various CPU/graphics combinations. A user-specified error threshold for pixel point sampling density and distance approximation governs the accuracy of the overall approach. Some of the novel features of our approach include:

- Improved and efficient construction of distance meshes used to compute 3D Voronoi diagrams accelerated with graphics hardware.
- Site culling algorithms and distance mesh culling for increased performance of Voronoi computation.
- Improved graphics hardware acceleration of computing the intersection between two, possibly non-convex, polygonal objects, over an entire volume.
- Improved algorithm for computing 3D image-space intersections that handles both inter-object and self-collisions.
- Computation of the gradients of the distance field using graphics hardware.

We have implemented our algorithm on various hardware configurations, and demonstrate its performance to compute different queries between rigid and dynamically deforming polygonal objects. Our approach is well suited for computing proximity query information needed for collision responses between dynamic deformable models. The use of graphics hardware allows us to perform different queries at interactive rates on complex deformable models. Moreover, it is relatively simple to implement all these queries in a robust manner. Over the last decade, the graphics processors (GPUs) processing power has been progressing at a rate faster than the CPUs and this will result in handling even more complex scenarios at interactive rates.

2. Related Work

Algorithms for computing collisions, intersections, and minimum separation distances have been extensively researched. Many are restricted to convex objects [Cameron 97, Ehmann01, Gilbert88, Lin91, Mirtich98] or are based on hierarchical bounding-volume or spatial data structures that require considerable precomputation and are best suited for rigid geometry [Hubbard93, Quinlan94, Gottschalk96, Johnson98, Klosowski98]. Some algorithms handle dynamically deforming geometry by assuming that motion is expressed as a closed form function of time [Snyder93] or by using very specialized algorithms [Baraff92]. In our approach, we emphasize the handling of non-convex, dynamically deformable objects with no precomputation or knowledge of object motions. In addition, we obtain computational complexity that grows linearly with geometric complexity for a fixed error tolerance and contact scenario.

As compared to collision detection and separation distance computation, there is relatively little work on penetration depth computation. Penetration depth is typically defined as the minimum translational distance needed to separate two objects. We define it with respect to a point as the minimum translational distance and direction needed to separate a penetrating point from an object's interior. Dobkin et al. have presented an algorithm to compute the intersection depth of convex polytopes, though no practical implementation is known [Dobkin93]. Cameron has presented a practical algorithm that computes an approximate depth for convex polytopes [Cameron97]. No practical algorithms are known for general, non-convex polyhedra.

Our algorithm relies on the computation of discretized distance fields and graphics hardware-accelerated geometric computation. Distance fields - scalar fields that specify minimum distance to a shape for all points in the field - have been used for many applications in graphics, robotics and manufacturing [Frisken00, Fisher01]. Common algorithms for distance field computation are based on level sets [Sethian96] or adaptive techniques [Frisken00]. However, they either require static geometry, extensive preprocessing, or lack tight error bounds. Graphics hardware has been used to accelerate a number of geometric computations, such as visualization of constructive solid geometry models [Goldfeather89], cross-sections and interferences [Rossignac92], and computation of the Minkowki sum [Kaul92]. However, these only compute intersections, not distance-related queries. Algorithms also exist for motion planning using graphics hardware acceleration and distance fields [Kimmel98, Lengyel90, Pisula00]. More recently, an algorithm has been proposed to compute generalized Voronoi diagrams and distance fields using graphics hardware [Hoff99]. Its application to motion planning was presented in [Pisula00]. Also, proximity queries accelerated using graphics hardware was presented in [Hoff01], but it was restricted to 2D and its extension to 3D was not obvious.

2.1 Voronoi and Distance field Computation

In [Hoff99], they present an algorithm for computing approximate 2D and 3D generalized Voronoi diagrams for polygonal objects with a variety of distance metrics. The representation is in the form of a discretized regular grid of sample points (images) across a 2D slice. A 3D Voronoi diagram is composed of a sequence of these slices across the volume to form a regular 3D grid. At each grid point, the ID of the nearest site and its associated distance is stored. They accelerate a brute-force algorithm using graphics hardware.

Instead of relying on a distance evaluation between a point and a Voronoi site, a polygonal distance mesh is constructed so that when rendered it computes the correct distance value as the Z-coordinate. If these distance meshes are rendered for each site with Z-buffer visibility enabled, the correct comparisons and updates will also be performed. This reduces the problem to finding a polygonal mesh approximation of a 2D slice of the distance function. In 3D, the distance mesh must approximate a 2D slice of the 3D domain.

Their 3D implementation simply used a coarse regular grid with direct distance evaluations at each grid point. This often required over-meshing, inefficient direct distance evaluations at grid points, and did not take advantage of the inherent symmetry in the functions being approximated. In addition, this approximation did not provide a tight bound on the approximation and the computation times were on the order of minutes to hours for high resolution Voronoi diagrams of complex models.

We extend the 3D distance mesh ideas and formulate a very fast and efficient bounded error approximation without requiring any lookup tables or complex data structures. In addition, we present methods for greatly accelerating the distance evaluations through culling techniques.

2.2 2D Proximity Queries using Graphics HW

In [Hoff01], they presented an approach using the graphics hardware based Voronoi computation for performing more general proximity queries, such as those needed in computing collision responses in a dynamics simulation. This paper focused on the interactions between 2D, possibly non-convex, polygonal objects only, but illustrated the potential for having a unified framework for a wide range of proximity queries. Many of the queries supported are particularly difficult for object-space algorithms, such as computing intersections, penetrating points, and penetration depths and directions. They used image-space techniques for performing these queries that were accelerated using graphics hardware. The core operations were based on queries into the Voronoi diagram. They presented a pipeline that allowed load balancing between CPU and graphics subsystems by first incorporating an object-space geometric localization phase to restrict the area over which the image-space phase must be performed.

Through improvements in the Voronoi diagram computation, we have extended this work into 3D. Many additional optimizations were necessary to make this run well in practice, including: faster and efficient distance meshing with bounded error, conservative Voronoi site culling, and making the queries symmetric (query A w.r.t. B is the same as B w.r.t. A). In addition, we constructed a specialized algorithm for computing 3D intersections efficiently. Previously in [Hoff01] for 2D, they relied on pixel overwrite to find intersection points. For 3D, we used a parity based strategy similar to operations used in graphics hardware-accelerated visualization of CSG operations and shadow volumes.

3. Overview of Our Approach

Given a collection of closed 3D polygonal objects, we perform coarse geometric localization to find rectangular regions of space (axis-aligned bounding boxes) that contain either potential intersections or closest feature pairs between objects. We uniformly point-sample these regions and use polygon rasterization hardware to compute object intersections, closest points, and the distance field. The distance field and its gradient vector field provide the distance and direction to the nearest feature for each point in the localized region, which gives the contact normals, minimum separation distances, or penetration depths. Our core algorithm computes the proximity information between two 3D, possibly non-convex, polygonal objects. Higher-order curved surfaces are tessellated into polygons with bounded distance deviation error. In our hybrid approach, there are two top-level operations:

(1) Geometric object-space operations to coarsely localize potential intersection regions or closest features

(2) Image-space operations using graphics hardware to compute the proximity information in the localized regions

Most of our improvements center around Voronoi and distance field computation since it is by far the most costly operation and is the most demanding of the graphics hardware. Load balancing between CPU and graphics subsystems is achieved by varying the coarseness of the object-space localization and by using object-space culling strategies. Tighter localized regions result in fewer pixels and a smaller bound on the maximum distance needed for Voronoi computation, thus reducing the fill and geometry loads on the graphics pipeline. We can also balance the load between these two main stages of the graphics pipeline by shifting the distance error tolerance in the Voronoi computation between fill and geometry: increasing the pixel resolution decreases the distance mesh resolution and vice versa. The main parts of the proximity query pipeline are shown in the following figure:

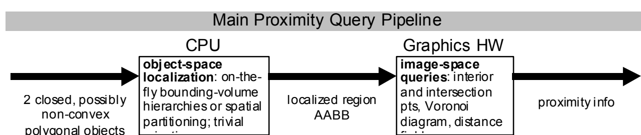


Figure 1: The proximity query pipeline is composed of two main stages: geometric localization and image-space queries. The most complex queries are performed by graphics hardware. Each stage can be varied to balance the load between CPU and graphics subsystems.

4. Object-space Geometric Localization

The image-based queries operate on a uniform 3D grid of sample points in regions of space containing potential interactions. The graphics hardware pixel framebuffer is used as a 2D slice of the grid and the proximity queries become pixel operations, therefore the performance varies dramatically with the pixel resolution. To avoid excessive load, a geometric localization step is used to localize regions of potential interaction or as a trivial rejection stage. This hybrid geometry/image-based approach helps balance the load between the CPU and graphics subsystems, giving us portability between different workstations with varying performance characteristics. More sophisticated geometric techniques, to tightly localize potential intersections or closest feature pairs, dramatically reduce the graphics pipeline overhead, but increases the CPU usage and the complexity of the algorithm. We use coarse fixed-height bounding-volume hierarchies to achieve this balance between speed and complexity, and between CPU and graphics usage.

There are many general and efficient algorithms available for localizing geometry based on bounding-volume hierarchies [Gottschalk96, Hubbard93, Johnson98, Quinlan94]. However, for exact collision detection these algorithms typically perform well only on static geometry where the hierarchy can be precomputed. In order to handle dynamic deformable geometry with no precomputation, we use coarse levels for efficient trivial rejection and obtain reasonable geometric localization. In addition, we perform lazy evaluation of relevant portions of the hierarchies while performing the collision or distance query. A subtree rooted at a particular node is only computed if its children need to be visited during the query traversal. The trees are destroyed after every proximity query, and no actual tree data structures are required since the child nodes are recursively passed to the query routine. A maximum height of each object tree is used to balance the CPU and graphics load. Similar algorithms can be constructed using spatial partitioning rather than bounding-volume hierarchies. Since the resulting localized region needs to be rectangular (an axis-aligned cube) to allow simple use of the graphics hardware, we use a dynamically constructed AABB-tree. With a fixed number (depth of the tree) of linear passes over the geometry we obtain reasonable localization.

The typical proximity query is between two objects at a time. However, it is possible to perform many simultaneous queries for all objects in an N-body simulation. We could perform the proximity queries for all objects with one image-space query by using a localized region that encloses the entire scene. This may be more efficient in cases when the objects are densely packed with many complex contacts throughout the space containing the objects. For example, in a dense rigid body simulation where many objects are interacting simultaneously (e.g. an asteroid field), a single image-space query over the entire space may be more appropriate (localized region is the world bounding box). In addition, as the computational power of graphics systems continues to overtake the general CPU power, coarser and simpler localization will be favored.

The geometric localization step may often result in multiple disconnected regions on each object. In these cases, the proximity query must be repeated for each localized region. Geometric localization for intersecting and nearest features can be found by using existing bounding-volume or spatial partitioning approaches that act on object boundaries, but finding localized regions around volume intersections requires a specialized algorithm. At each step of refinement, the parent bounding box must fully contain the volume intersection. This can be accomplished by first starting with the intersection of the top-level object bounding boxes. This intersection box will surely contain the intersection volume. Now we can refine this localization by computing the bounding box of the portion of each object that lies in the current box. We then repeat the process on the intersection of these two boxes which is also guaranteed to contain the intersection volume.

5. Image-space Proximity Queries

The proximity queries are simplified using uniform point sampling inside an axis-aligned bounding box (localized region) and accelerated with graphics hardware. This image-space approach helps decouple the objects' geometric complexity from the computational complexity for a specified error tolerance. We point-sample the space containing the geometry within the localized regions with a uniform rectangular 3D grid and perform the queries on this volumetric representation using graphics hardware acceleration. The image-based queries include computing intersections between objects, computing the distance field of an object boundary, and computing the gradient of the distance field. Variations of these basic operations are used to perform the remaining queries. The basic pipeline is shown in Figure 2.

The 3D image-space queries avoid excessive data handling when processing the entire volume of the localized region. Each query must be performed over the uniform 3D grid, one 2D slice at a time. The application query information is sent to the application as it is processed slice-by-slice to avoid processing and storing the entire 3D image. In addition, many of the queries have been made symmetric to avoid a second pass as needed in the previous work.

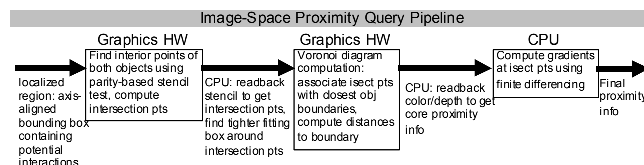


Figure 2: The most computationally intensive tasks are performed by the graphics hardware. These stages are also the most difficult for geometric object-space approaches. We accelerate simple brute-force image-space solutions using graphics hardware to obtain interactive performance on complex models with no precomputation

5.1 Intersections

We compute intersection points on a 2D slice by performing a parity test, as is often used in shadow volumes and CSG rendering, using graphics hardware stencil operations [Crow77, Rossignac92]. In order to find intersections, we must first be able to identify sample points that are inside the object. The set of sample points that are inside both objects form the intersection points between the objects. We then describe another generalized strategy that can handle intersections between multiple objects simultaneously along with the more complex self-intersections.

For any closed object, we can determine if a point is inside the object by shooting a ray from the point in any direction and counting the number of times the object's surface is crossed. If the count is even, the point is outside of the object; if odd, the point is inside. We can simultaneously determine which sample points on a 2D planar slice are interior points by projecting all of the geometry on one side of the plane onto the plane and counting the number of times pixels are overwritten. This computation is performed using the graphics hardware through an odd-even parity test for rendered geometry clipped by the plane and projected onto the plane. Each time a pixel is overwritten the parity bit is flipped. Pixels whose stencil bit is set to 1

represent points on the slice that are inside the object. Initially the stencil buffer is initialized to 0.

5.1.1 Incremental Update and Bucket Sorting

For a single slice, this computation requires rendering all of the geometry on one side of the plane (clipped by the plane). However, this is inefficient for evaluating interior points on many slices swept through our 3D localization box. We improve efficiency by performing a plane sweep and updating the stencil buffer incrementally. For each slice, we only render the geometry between the current slice and the previous slice.

This incremental update improves the running time dramatically since on average the entire model is only drawn once! As opposed to the single slice approach where the entire model to one side of the slice had to be drawn for each successive slice. We can obtain even greater performance by first sorting the geometric primitives along the Z-axis by their minimum Z-values. A general sort would require O(n log n) time complexity. We obtain expected O(n) complexity by performing a bucket sort by using the slab positions as the buckets. With one pass through the geometry, we can assign each primitive to a bucket by its minimum Z-value. We maintain a list of currently active geometry for each slab. For each subsequent slab we add geometry to the list from the associated bucket. Geometry is removed from the list by checking if the old primitives' max Z-value is less than the current slab NearZ (swept past the primitives). This also dramatically improves performance with very little extra complexity or data. We avoid having to search for geometry that intersects the current slab. In addition, there is no need to add geometry to the buckets that lies outside of the XY min/max box. In practice, very little geometry has to be processed for the interior computation.

In order to find the intersection between two objects, we compute the interior of both objects inside of the localized region one slice at a time. The interior of both objects is encoded in a different bit of the stencil buffer. The set of points with both bits set are intersection points since they are interior to both objects. To actually extract these points, we must read the stencil image and search for the pixels with the appropriate value (a value of 3 from the 1st and 2nd least significant bits being set). These points must then be transformed from pixel-space into object-space.

5.1.2 Complexity and Error Analysis

Our new algorithm for intersection computation for 3D non-convex objects is simpler as compared to the 2D intersection computation algorithm presented in [Hoff01]. The major weakness of finding overwritten pixels between two non-convex polygons, was that they had to be triangulated in order to be rendered. This was the dominant part of the intersection computation since it was worst case O(n log n) rather than O(n). However, the expected running time of most triangulation algorithms is usually close to linear. In 3D, we only require the O(n) complexity where n is the number of primitives. The actual running time varies most dramatically with the ratio of the size of the localized region over the error tolerance, and is largely independent of the geometric complexity. More complex forms of contact do not result in increased running times unless the size of the localized region is increased dramatically or the error tolerance is reduced. These cases are difficult to analyze since they vary dramatically with the object configurations. More sophisticated geometric localization will reduce performance variations.

The complexity of rendering objects grows linearly with respect to the number of primitives for a fixed pixel resolution. Computing intersections geometrically between two polygon boundaries is worst case $O(n^2)$ since all primitives could intersect each other. The complexity of our algorithm is O(n) where n is the number of primitives. The hierarchical geometric localization step is also O(n) since the maximum depth of the tree is held constant. This tree depth balances the load between the CPU and graphics subsystems.

Similarly to the 2D case, the error in the interior and intersection computation is related to the pixel error in scan-conversion. The actual interior regions will never be off by more than half of the length of the diagonal of a pixel's rectangular cell (the error tolerance). The error tolerance has a dramatic effect on the number of pixels that have to be processed. When reduced error tolerances are required, better geometric object-space localization must be employed to reduce the load on the graphics subsystem. Furthermore, we can also balance the loads between geometry and fill stages of the graphics pipeline by trading off error in the pixel resolution and the distance mesh granularity.

Incorrect intersection parity resulting from pixel sample points lying exactly on tangent points to the object surface are avoided through correct minimum-based triangle rasterization as described in [Rossignac92]: either the crossing will be counted twice or not at all.

5.1.3 Multiple Objects and Self-Collisions

We can modify the intersection routine to handle self-collisions and multiple objects with very little modification to the previous algorithm. The modification adds the complexity of having to distinguish front and back faces for polygons in each slab for a parallel projection and has the slight restriction of only handling the intersection of at most 255 simultaneous volumes (limit of 8-bit stencil buffer).

Instead of finding the interior of both objects separately and then finding their common intersection, we can simply finding the intersection directly using the geometry from both objects simultaneously using the classic parity test used in the shadow volume algorithm. Since we want to know if a point is inside two volumes simultaneously, a ray emanating from a query point must have exited at least two more volumes than it has entered.

Instead of simply flipping a bit each time a boundary is crossed (front or back facing), starting with a stencil counter initialized to zero, we increment the counter each time a volume is exited (a back face is rendered) and decrement the counter whenever a volume is entered (front face is rendered). The counter will indicate the number of objects containing the point. We are interested in the intersection points, so the counter must at least be 2. We simply modify our existing approach of rendering slabs to perform this count instead. We must classify all object faces for each slab as front or back facing with respect to a parallel projection. Since all object triangles are handled together, we can handle more than 2 objects and we can also find self-intersections of a single object. Stencil counts of 2 or greater indicates a point that is in the intersection of at least one pair of objects or an object with itself.

5.2 Distance Field Computation

We use the algorithm presented in [Hoff99] for constructing generalized Voronoi diagrams using graphics hardware for 3D polygonal objects. This approach computes an image-based representation of the Voronoi diagram in both the color and the depth buffers for one 2D slice of the 3D volume at a time. A pixel's color identifies the polygon feature (vertex or edge) that is closest to the slice pixel's sample points; its depth value corresponds to the distance to the nearest feature. The depth buffer is an image-based representation of the distance field of the object boundaries. The distance field is computed by rendering 3D bounded-error polygonal mesh approximations of a 2D planar slice of the distance function where the depth of the rendered mesh at a particular pixel location corresponds to the distance to the nearest geometric feature.

The goal in constructing a distance mesh is to find a piecewise linear approximation across a 2D planar domain of a Voronoi site's 3D

scalar distance function. The distance to a site from a point (x,y,z) is defined as $D(x,y,z)$. The function we are interested in approximating is for a 2D planar slice $z=Z_{slice}$. So we wish approximate the 2D scalar function $D(x,y, Z_{slice})$, where Z_{slice} is a constant for any particular slice, such that the approximation D' and actual distance function D never differ by more than the user-specified distance error. In addition, the domain across the slice is restricted to a 2D window and the range of the function is restricted to $z\in[0,MaxDist]$. The shape of the distance mesh for a 3D point is one sheet of a hyperboloid of two sheets; for a line, an elliptical cone; and for a plane, another plane.

In [Hoff99], distance meshes were constructed using lookup tables. We construct error-bounded polygonal mesh approximations of a 2D planar slice of a primitives distance function at run-time with no precomputation at faster rates than the algorithm based on lookup tables. We solve for the mesh stepsizes needed to maintain the desired error threshold while taking advantage of symmetry. We attempt to actual obtain the desired error to make the meshes as coarse as possible for rendering efficiency. In addition, we only construct geometry that lies within the slice window.

For computing distance fields for proximity queries, we obtain higher performance than the generalized Voronoi diagram computation because of the localized regions. In the case of computing distance fields for proximity queries, the localized regions always contain the geometry that is in potential contact or that contains the closest features. The farthest away points on two objects can be is in opposite corners of the localized region box. So the maximum distance we need to construct distance meshes for is half of the diagonal length of the box. Reducing the maximum distance results in the greatest speedups in Voronoi computation since it reduces geometry and fill by reducing the overall extent of the distance meshes, and the smaller bound allows the objects to be easily culled if they are too far from the localized region thus avoiding distance mesh construction completely. In addition, the distance mesh generation routines attempt to minimize the number of primitives drawn by constructing a mesh that is as coarse as possible while staying within the specified error bound (the error bound is tight, this deviation can actually be measured for various places in the mesh approximation) and by only generating primitives that are inside the localized region bounding box. In addition, in many proximity queries we can further reduce the maximum distance needed when we only want intersection or closest points near the boundaries of the object.

5.3 Gradient of the Distance Field

We compute the gradient of the distance field at pixel locations by using central differences in all three principal axis directions. In practice, this simple approach gives reasonable results even with the distance error and lack of C^1 or higher continuity in the polygonal distance mesh approximations used to compute the distance field. Gradients are computed in software for selected points after reading back the distance values. If the entire gradient field is desired, we could accelerate the computation using multi-pass rendering or pixel shading operations.

The most difficult problem in computing the gradient is in handling discontinuities and boundaries in the distance field. There are three types of discontinuities that occur: across Voronoi boundaries, across Voronoi sites, and at the boundaries of the grid. In each case, the support of the finite differencing "kernel" has to cross a discontinuity and gives an incorrect gradient. A more robust method is shown in the fast marching methods in [Sethian96]. He solves for a distance value at an unknown point using an implicit method based on the fact that at least one adjacent distance value must be known and does not cross a discontinuity, and that for the nearest Euclidean distance metric the magnitude of the gradient must be 1 everywhere (except at the discontinuities). We use the same method by just using the one-sided difference in each direction that will result in a gradient whose magnitude is 1 (choose the adjacent value in each direction that has the maximum difference). Adjacent distance values that cross a discontinuity will not be chosen. An alternative, but slightly more complex, strategy is to compute the gradients of the continuous distance meshes directly.

By directly encoding gradients at distance mesh vertices, we can use the linear interpolation of polygon rasterization to compute gradients at all pixels. Since we would be linearly interpolating a gradient, this gives us a higher order interpolation than central differencing of adjacent distance values. This is comparable to the difference between Gouraud and Phong interpolation (the first linearly interpolates color values across a polygon, the second linearly interpolates the surface normal for per-pixel lighting calculations). In addition, the gradient is much simpler if computed only for a single site at a time during distance mesh construction. We need only provide the direction to the nearest point on the site at each distance mesh vertex. The main difficulty with this approach is in the encoding of the gradient for rapid computation by graphics hardware.

This approach as some difficulties due to limitations of graphics hardware framebuffer precision. There are a number of ways we can interpolate the gradient information. The simplest is to encode the signed normalized components into the 8-bit RGB color values at each vertex (using hardware scale and bias operations for sign). The linear interpolation would give the correct results to 8-bits of precision. This approach introduces quantization error when encoding and additional error during interpolation. Using 3D texture coordinates, high precision encoding and interpolation is obtained. However, the resulting per-pixel texture coordinates are still quantized to low precision RGB values in the framebuffer. The texture-mapping function would simply be the identity mapping. We are interpolating (s,t,r) gradient values and we want those values directly at each pixel. The graphics hardware does not allow higher precision intermediate results for multi-pass operations. However, the texture-mapping method has the advantage of only introducing significant error at the final stage; encoding and interpolation are done at floating point precision. Also, the signs will be correctly handled without any additional scaling or biasing. However, we also have no simple way of performing the identity map. We must use a 1D texture that maps [-1,1] to [0,255], but this can only be applied to one texture component at a time. This would require three passes in order to transform (s,t,r) into RGB values. A less efficient approach would involve the use of a 3D texture map. Current pixel-level programmable graphics hardware may provide a simpler and more efficient way to handle this mapping.

5.4 Other Proximity Queries

We use the basic operations of computing interior points, intersections, the distance field, and the gradient of the distance field to perform the other proximity queries mentioned in section 1.

Penetration Depth and Direction: For a point on object A that is penetrating object B, we define the penetration depth and direction for the point as the distance and direction to the nearest feature on B. This information is provided directly from the distance field and its gradient computed at the penetrating point. Penetrating points are found using the intersection operation. Intersection points are associated with each object based on the Voronoi diagram's color buffer that indicates the closest object to each point. Contact points and Normals are computed in the same way. Approximate contact points result from the objects slightly penetrating each other.

Closest Point: We find the point on object A that is closest to object B by first geometrically localizing potential closest feature regions (one bounding box on each object) using some hierarchical approach. We then compute the distance field of object B and the interior points of A in A's localized region (gives us minimum distance to B for all points in A in A's localized region). We then search these points to find the one with the smallest distance value. This point will be the point on A

that is closest to B. This process has to be repeated for B with respect to A. This requires two passes, but the interior points and the distance field needs to only be computed once for each object.

Separation Distance and Direction: We find the minimum separation distance and direction between two objects A and B by first computing the closest point on A to B and vice versa. Ideally, we find the closest point on B to A from the distance value and gradient at the closest point on A to B, but the amplification of errors over the greater distance may cause problems. The distance between these two closest points is the separation distance and the line segment between them gives the separation direction.

6 Performance

We tested the system performance in both rigid and deformable body dynamic simulations on a several different hardware configurations. In the rigid body cases, we measured the performance of the system in computing proximity query information needed for computing a penalty-based collision response. In these cases, only shallow penetration is allowed since the objects bounce off of each other. For the deformable cases, we perform only the proximity queries without collision response to show the worst case of computing proximity information for many deep simultaneous contact scenarios with dynamically deforming geometry. We tried to choose three hardware configurations that would reflect variations in balance between CPU and graphics computational power:

(1) Pentium-4 1.8Ghz with GeForce3 Ti500 graphics: fast CPU and fast graphics

(2) 1 graphics pipe and 1 300Mhz MIPS R12000 processor of an SGI Reality Monster with InfiniteReality2 graphics: slower CPU and fast graphics

(3) PentiumIII-750Mhz laptop with ATI Rage Pro LT: fast CPU and slow graphics.

Because of the ability to balance the load between the CPU and graphics subsystems and between stages of the graphics pipeline, we are able to achieve interactive performance on all configurations. In most cases, we only needed very simple one-level geometric localization (intersection of top-level axis-aligned bounding boxes). Most of the balancing was between stages of the graphics pipeline (much of the geometry stage on older graphics systems was performed on the CPU: before hardware T&L). We also show the effects of the varying the distance threshold on system performance.

For performance evaluation, we implemented a rigid body simulator with collision response and a variety of deformable simulations without collision responses to allow more complex contact scenarios. The test scenarios vary from simple convex objects composed of around 2 thousand triangles with simple contact regions to non-convex objects with nearly 10,000 triangles with multiple complex overlapping and interlocking contact regions. The average query times are shown in Table 1. It is important to note that the query time is not growing because of the increase in geometric complexity, but rather because our more complex models are in more complex contact configurations.

The performance of our image-space query system depends more on the contact configuration than on the complexity of the objects. The distance error tolerance determines the point sample density across the contact volume. The density and the volume of the localized regions and the contact regions determine the number of pixels that have to be processed. If an insufficient level of geometric localization is used, the number of pixels to process may increase dramatically. The user must decide the appropriate amount of localization to properly balance the CPU/graphics load. In addition, the performance can be varied dramatically by the user-specified distance error tolerance. In Table 2, we show the effects on performance with a varying error tolerance.

Average Total Per-frame Proximity Query Times					
Demo	**#Tris**	**Isect Pts**	**GeForce3**	**IR2**	**Rage Pro LT**
Cylinders	2000	513	12ms	61ms	45ms
Tori	5000	1412	71	262	257
Heart	8000	317	149	329	434
Rigid	15000	2537	313	1001	966

Table 1: Performance timings for dynamics simulations. The number of triangles, average number of intersection points, and average time to run proximity queries per frame is reported for error tolerance d (see Table2). Timing data was gather from three machines, a Pentium4 1.8GHz desktop with a 64Mb GeForce3, a SGI 300MHz R12000 with InfiniteReality2 graphics, and a Pentium-III 750Mhz laptop with ATI Rage Pro LT graphics.

Effects of Error Tolerance on Performance				
Error	**Isect Pts/Frame**	**GeForce3**	**IR2**	**Rage Pro LT**
***d*/4**	89605	548ms	1701ms	2846ms
***d*/2**	11238	169	578	689
d	1413	71	262	257
2*d*	177	32	189	103
4*d*	22	15	56	40

Table 2: The effect on performance when changing the distance error tolerance *d*. The average number of intersection points per frame is also reported. We used proximity queries on the deformable tori demo. The error determines the number of pixels used in the image-based operations. Systems with low graphics performance are more directly affected by the choice of *d*; however, as the error is increased there is less dependence on graphics performance and the faster laptop CPU overtakes the InfiniteReality2 system.

Although we focused most of our efforts on handling deformable body proximity queries, our system is also applicable to rigid body queries. We use a penalty-based collision response that acts on individual point samples that approximate our object. These point samples arise from our image-space proximity queries. Particles are allowed to penetrate objects in penalty-based collision response computation. When a penetration is detected, a spring based restoring force, whose magnitude is proportional to penetration depth, is then applied to the particle until it has separated from the object. The measure of penetration is notoriously expensive to compute and limits the use of penalty-based techniques to mostly models decomposable into convex primitives. The generality and computational efficiency provided by our proximity query algorithms alleviates this problem.

7 Conclusion and Future Work

We have presented a hybrid geometry- and image-based algorithm for computing geometric proximity queries between two non-convex closed 3D polygonal objects using graphics hardware. This approach has a number of advantages over previous approaches. The unified framework allows us to compute all the queries, including penetration depth and direction and contact normals. Furthermore, it involves no precomputation and handles non-convex objects; as a result, it is also applicable to dynamic or deformable geometric primitives. In practice, we have found the algorithm to be simple to implement (as compared to similarly robust geometric algorithms), quite robust, fast (considering the complexity of the queries), and very flexible. We have developed an interactive system that shows proximity queries computed between 3D dynamic deformable objects to illustrate the effectiveness of our approach.

References

[Baraff92] D. Baraff, *Dynamic Simulation of Non-Penetrating Rigid Bodies*. Ph.D. Thesis, Dep of Comp. Sci., Cornell University, March 1992

[Cameron97] S. Cameron, *Enhancing GJK: Computing Minimum and Penetration Distance between Convex Polyhedra*. International Conference on Robotics and Automation, 3112-3117, 1997

[Crow77] F. Crow, *Shadow Algorithms for Computer Graphics*. SIGGRAPH 77.

[Dobkin93] D. Dobkin, J. Hershberger, D. Kirkpatrick, S. Suri, *Computing the Intersection Depth of Polyhedra. Algorithmica*, 9(6), 518-533, 1993

[Ehmann01] S. Ehmann and M. Lin. *Accurate and Fast Proximity Queries between Polyhedra Using Surface Decomposition*. Eurographics 2001

[Fisher01] S. Fisher and M. Lin. *Fast Penetration Depth Estimation for Elastic Bodies Using Deformed Distance Fields*. Proc. Intl. Conf. on Intelligent Robots and Systems, 2001

[Frisken00] S. Frisken, R. N. Perry, A. P. Rockwood, T. R. Jones, *Adaptively Sampled Distance Fields: A General Representation of Shapes for Computer Graphics*. SIGGRAPH00, 249-254, July 2000

[Gilbert88] E. G. Gilbert, D. W. Johnson, S.S. Keerthi. *A Fast Procedure for Computing the Distance Between Objects in Three-Dimensional Space*. IEEE J. Robotics and Automation, RA(4): 193-203, 1988

[Goldfeather89] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs. *Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning*. IEEE Computer Graphics and Applications, 9(3):20-28, May 1989

[Gottschalk96] S. Gottschalk, M. C. Lin, D. Manocha, *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection*. SIGGRAPH 96, 171-180, 1996

[Hoff99] K. Hoff, T. Culver, J. Keyser, M. Lin, and D. Manocha. *Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware*. SIGGRAPH 99, 277-285, 1999

[Hoff01] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha. *Fast and Simple 2D Geometry Proximity Queries Using Graphics Hardware*. ACM Symposium on Interactive 3D Graphics, 2001

[Hubbard93] P. M. Hubbard, *Interactive Collision Detection*. IEEE Symposium on Research Frontiers in Virtual Reality. 24-31, 1993

[Kaul92] A. Kaul and J. Rossignac, *Solid-interpolating Deformations: Construction and Animation of PIPs*, Computer and Graphics, vol 16, 107-116, 1992

[Kimmel98] R. Kimmel, N. Kiryati, A. Bruckstein, *Multi-Valued Distance Maps for Motion Planning on Surfaces with Moving Obstacles*. IEEE Transactions on Robotics and Automation, vol 14: 427-438, 1998

[Klosowski98] J. Klosowski, M. Held, J. Mitchell, K. Zikan, H. Sowizral. *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*. IEEE Trans. Vis. Comp. Graph, 4(1):21-36, 1998

[Johnson98] D. Johnson, E. Cohen, *A Framework for Efficient Minimum Distance Computation*, IEEE Conf. On Robotics and Animation, 3678-3683, 1998

[Lengyel90] J. Lengyel, M. Reichert, B.R. Donald, and D.P. Greenberg. *Real-time Robot Motion Planning Using Rasterizing Computer Graphics Hardware*. Computer Graphics (SIGGRAPH 90 Proc.), vol. 24, pgs 327-335, Aug 1990

[Lin91] M. Lin, J. Canny. *Efficient Algorithms for Incremental Distance Computation*. IEEE Transactions on Robotics and Automation, 1991

[Mirtich96] B. Mirtich, *Impulse-Based Dynamic Simulation of Rigid Body Systems*. Ph.D. Thesis, University of California, Berkeley, Dec 1996

[Mirtich98] B. Mirtich, *V-Clip: Fast and Robust Polyhedral Collision Detection*. ACM Trans. on Graph, 17(3):177-208, 1998

[Pisula00] C. Pisula, K. Hoff, M. Lin, and D. Manocha. *Randomized Path Planning for a Rigid Body Based on Hardware Accelerated Voronoi Sampling*. Proc. of Workshop on Algorithmic Foundations of Robotics, 2000

[Quinlan94] S. Quinlan, *Efficient Distance Computation between Non-Convex Objects*. International Conf. on Robotics and Automation, 3324-3329, 1994

[Rossignac92] J. Rossignac, A. Megahed, and B. Schneider. *Interactive Inspection of Solids: Cross-sections and Interferences*. SIGGRAPH 92, 26, 353-360, July 1992

[Sethian96] J. Sethian, *Level Set Methods*, Cambridge University Press, 1996

[Snyder93] J. Snyder, A. Woodbury, K. Fleischer, B. Currin, A. Barr, *Interval Methods for Multi-Point Collisions Between Time Dependent Curved Surfaces*. ACM Computer Graphics, 321-334, 1993

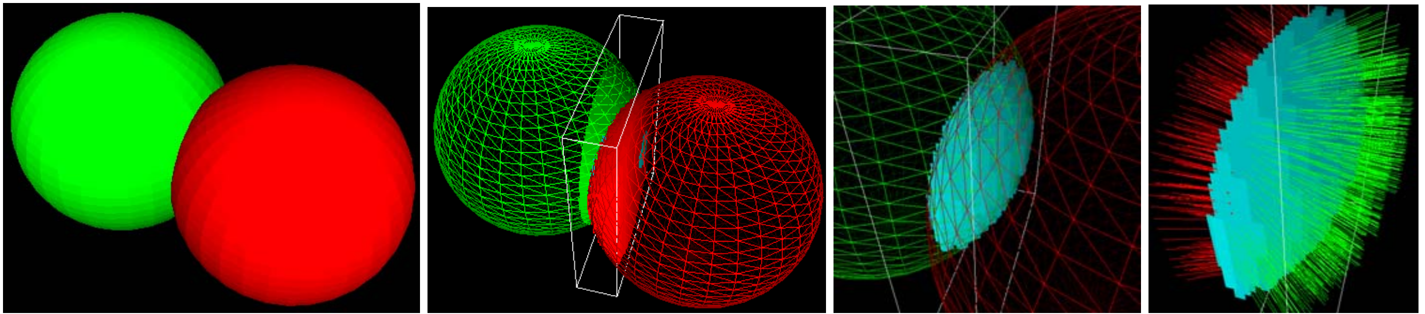
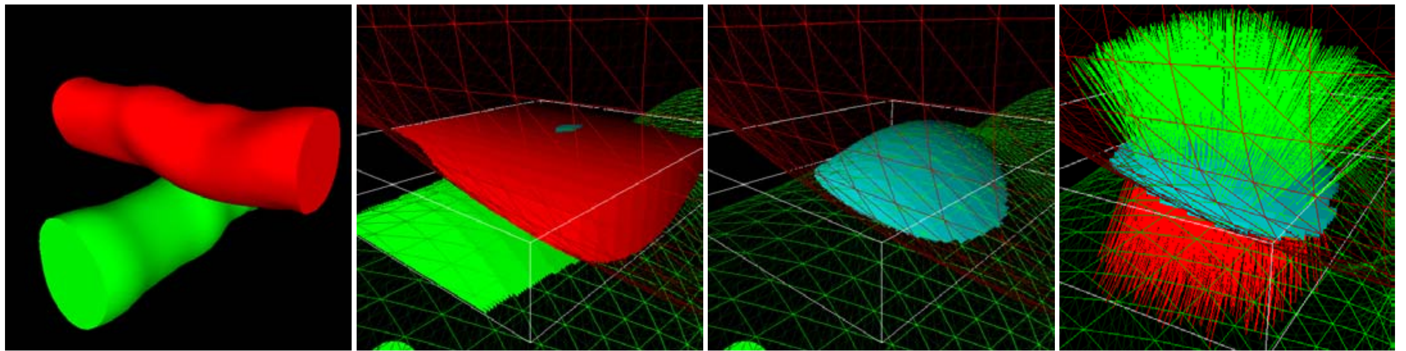


Plate 1 **hybrid proximity query pipeline**: Given two closed polygonal objects, a coarse object-space geometric localization step is performed to find an axis-aligned bounding box that contains a potential interaction (2). Inside the localized region, the lower-level image-space queries are performed. First the interior of each object is indentified using an incremental stencil parity test for a series of 2D slices across the volume (2). The set of point that are determined to lie in the interior of both objects form the intersection points between the objects (3). Then, the Voronoi diagram is computed inside a tighter region around the intersection points at the same resolution as the intersection resolution. The Voronoi diagram serves two purposes: associates intersection points with their closest object boundaries, and provides the distance field. The distance value at an intersection point gives the penetration depth, and the gradient gives the penetration direction.



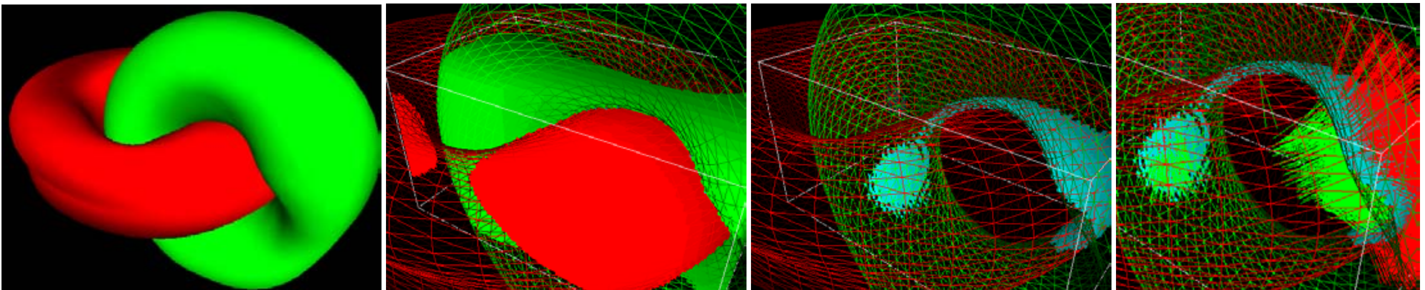


Plate 2 **real-time dynamic deformable proximity queries**: The same proximity query pipeline can be applied to dynamic deformable models where every vertex is assumed to change for every frame. The complex contacts between non-convex objects can result in disconnected intersection regions. Each cylinder model is composed of 2000 triangles and the average query time is 12ms for an average of 513 intersection points per query. The tori are composed of 5000 triangles and the query time is 71ms for 1412 intersection points. Each simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

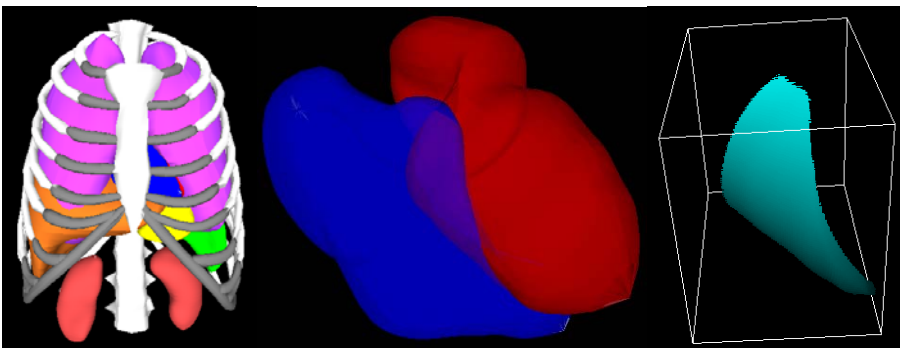


Plate 3 **proximity queries on body heartbeat simulation**: The proximity queries are used for path verification of the organs during a precomputed breathing simulation. Here we can see that the two ventricles are actually intersecting. The heart is composed of 8000 triangles and the average query time is 149ms for an average of 317 intersection points. This simulation performed at interactive rates on a Pentium4 1.8GHz desktop with a 64Mb GeForce3.

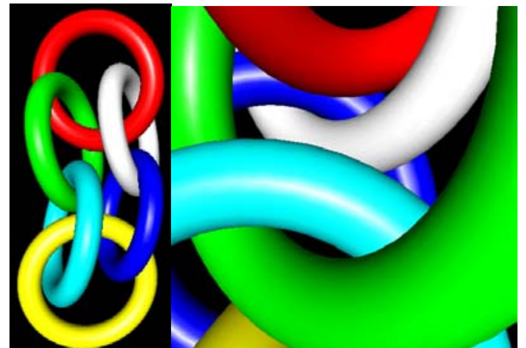


Plate 4 **multiple complex contact scenario in an interactive rigid body simulation**: Collision responses are computed using a penalty-based method that requires penetration depth computation. Each ring is composed of 2500 triangles, average query time is 313ms for 2537 intersection points.